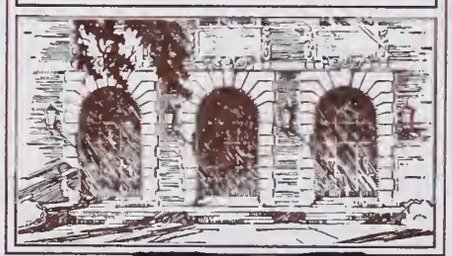


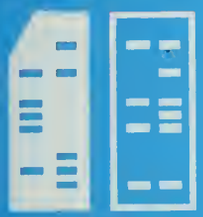
UIUCDCS-R-76-831

CHARACTERIZATION OF A DISTRIBUTED DATA BASE SYSTEM

BY

ENRIQUE GRAPA

October, 1976



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-76-831

CHARACTERIZATION OF A DISTRIBUTED DATA BASE SYSTEM

BY

ENRIQUE GRAPA

October 1976

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

This work was supported in part by the Department of Computer Science, the Center for Advanced Compuation and the Command and Control Technical Center and was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science, at the University of Illinois.

ACKNOWLEDGMENTS

I am greatly indebted to Geneva Belford for her guidance and unmeasurable patience during the preparation of this thesis.

My sincere appreciation to Professor Dan Slotnick for his continuous encouragement and for his revision of the different stages of this work.

I would like to express my appreciation to Professor Peter Alsberg for reviewing this thesis, and for making it possible by accepting me as a member of his research group.

I am very grateful to the directorate of the Instituto de Investigacion en Matematicas Aplicadas y Sistemas (IIMAS) of the National University of Mexico (UNAM) for opening the doors to my graduate studies by commissioning me to this University.

Dr. Renato Iturriaga's advice and encouragement during the early stages of my graduate studies is very deeply and warmly appreciated.

My unrestricted thanks go to Pamela Hibdon for the excellent and careful typing job that she performed and to Greg Ives who made all the drawings.

Last, but by no means least, I am very deeply indebted to my mother, friends and relatives for their patience and support, especially to my wife Rebeca and children Arie and Nurit whose mere existence and love fuels my motivation.

TABLE OF CONTENTS

Chapter I. INTRODUCTION

The literature is full of papers describing the advantages of distributed data base systems. Studies to determine the location of different copies of such a data base are abundant as well. All of these are based upon models with unproven workability. Strangely enough, no distributed data base system is available and efforts to determine the building parameters of such a system have been feeble. The main goal of this thesis is to obtain a characterization of a workable distributed data base system in order to provide appropriate tools for a future implementation.

The literature uses the term distributed data base system for different concepts. It is thus necessary to clearly define the phrase. For the purpose of this thesis, a distributed data base system is characterized by:

a) multiple copies of portions of the data base distributed among various sites of a computer network;

b) geographic separation of the participating computer systems with a limited communication bandwidth between them; and

c) co-equal roles for the systems involved (i.e., similar software at each computer rather than a single piece of software distributed throughout the network).

Distributed data base systems use a network as the underlying hardware facility rather than a single computer. This difference impacts the behavior of a data base system. In a network it can easily be the case that two or more copies of a data base are cheaper than a single copy. By going to multiple copies we can reduce query transmission cost. This saving can offset the additional costs incurred by the additional copy (storage, etc.).

Related literature occurs in several areas: work on optimal location, work on multiple copies, or work on both combined. In the first category are papers (such as [A1]) dealing with the determination of the optimal site for a given task. In the second category ([C5], [B2]) are studies indicating the advantages of multiple copies over single copies according to some specific criteria (e.g. availability). Both issues are combined in the file allocation problem. In the file allocation literature [e.g., C1, C2, C3, C4, etc.] we find that the need for multiple copies is used to justify the search for an optimal number and location of copies of a data base.

Alsberg [A1] has shown that in heterogenous networks (specifically in the ARPA network) it is quite possible that substantial savings might be obtained by using the proper computer; i.e., that in many situations transmission costs can be more than offset by the savings in time and money produced by the use of the right computer. This opens the door to additional possible savings in a distributed data base system by maintaining different organizations or data at each site [A2].

Even without an economic rationale, a distributed data base system could be dictated by requirements imposed on the system's design. In many applications availability or response time are design constraints. Putting a single copy of a data base at the best site might improve availability and response, but only up to a certain point. Beyond that point distributed data base systems (including multiple copies) are required.

In the file allocation literature (e.g. [C1]) we find various examples in which multiple copies could produce transmission cost savings (see section II.2) that will offset the additional cost of an extra copy. Thus, in a search for the system with the lowest operational cost, we might obtain an unexpected

surprise. It could well be that besides improving costs we will improve availability and response time as well.

Considering these comments, we should expect a literature full of specifications for working distributed data base systems; this is hardly the case. There exists a fair number of papers describing different ways in which copies of a file can be allocated in the network to improve operational costs [C1, C2, C3, C4, C5, L2, S1, U1]. However the emphasis of these papers is on the mathematical side. The authors fail to justify the workability of the distributed data base models they use. Casey [C1], for example, assumes that every site is a generator of updates, which are broadcast to all sites that maintain a copy of the data base. In general, this approach affects seriously the consistency and validity of data in the different data base copies and should be either rejected or modified. The literature shows that this has not been done. Moreover, Casey's work has triggered some extensions [L2, U1] based on the same unsafe approach.

Only one research group outside the University of Illinois has studied the actual implementation of a distributed data base system. Johnson and coworkers [J1, J2] have presented a model (see chapter III) that attacks the synchronization problems. The discussion in chapters IV, V and VII will indicate the limitations of this model.

Chapter II will be more specific about the advantages of a distributed data base system. It will cover the available literature that helps to support the usefulness of such a system. Chapter III will discuss the available models for a distributed data base system, including one of our own (the Reservation Center). Chapter IV will evaluate the pros and cons of these models and will pinpoint some flaws. The major flaws will be discussed in chapter V, in which modifications of the models to solve some of the problems will be presented. In

chapter VI we will return to the file allocation problem and make an a posteriori evaluation of the validity of available models. The presentation of some personal contributions in this area will be shared between this chapter and the appendices. Finally, chapter VII will present our conclusions.

The numbering scheme for figures, tables and formulas includes the section number in which they appear as a prefix. I.e., figure II.3-a can be found in section II.3, etc. Figures have letters of the alphabet for a suffix. Tables use arabic numbers. Formulas are indicated by small roman numbers. (For example, we refer to table II.2-1, figure III.1-c, and expression II.3-ii.)

Chapter II. WHY IS A DISTRIBUTED DATA BASE SYSTEM USEFUL?

II.1 Introduction

In previous comments, we have mentioned that a distributed data base system could be cheaper and better from the standpoint of availability and response time. In this chapter the available literature is reviewed. This helps to stress the potential usefulness of a distributed data base system. At the same time, this review puts into proper perspective such subjective terms as "cheaper" and "better". For example, "cheaper" is shown to be related only to operational costs. Research and implementation costs (which are clearly nontrivial, since no distributed data base system actually exists) are completely neglected.

This chapter is divided into three technical sections:

II.2 network economies,

II.3 availability, and

II.4 response times.

In section II.2 we will discuss the economies that could be obtained by the proper use of network facilities. An important part of this section will be the consideration of the file allocation problem; i.e., where to allocate copies of a file in order to minimize operational costs. In the last two sections we will refer to the existing literature to show clearly how availability and response time are improved by having multiple copies.

II.2 Network economies

A distributed data base system could offer interesting economies by capitalizing on resource variety and/or by the proper exploitation of the topology of the network.

Our first point is justified by Alsberg's work [A1]. He made an experiment in which different types of programs were submitted to the different facilities in the ARPA Network. The result was that there was often a cost difference of one or two orders of magnitude between the best and the worst computer system. Furthermore, the roles of best and worst shifted as the type of program was changed. All the systems studied were among the best for some types of programs and among the worst for others. This result strongly motivates the study of optimal locations for a given data base, since savings can obviously be obtained. Moreover, if we have a multiple copy distributed data base system, there is no restriction on the way the information is locally organized. Thus similar savings could be obtained by relaying a data base operation (query) to the site with the best organization to treat it [A2]. Considering the potential that this area offers, we find it surprising that, as far as we know, no additional work has been done. This deficiency might be explained by the specificity of the required research. That is, one has to get involved with specific models of computers in a specific network. The multiplicity of models and operating systems makes this area not very attractive for research.

Another way of profiting from the existence of a variety of resources is by load sharing. An overloaded computer could share part of its load with an underutilized one. This generally improves the overall throughput of the network. However, the effect of load sharing in network economies is indirect. Transferring a job to a second computer with unknown qualities does not guarantee a cheaper cost; quite the contrary. However, the best utilization of the network could permit higher throughput rates and probably cause lower cost per unit of service. Further discussion of this topic is given in section II.4.

A second source of network economies is the exploitation of the topology of the network. The network may have sites which have different query

and update requirements for the use of a given file. It is clear that the topology will be important to the decision as to where file copies are placed and how many copies are needed for maximum economy. This problem, known as the file allocation problem, has been studied by various researchers [e.g., C1, C2, C3, C5, L2].

The remaining part of this section will be dedicated to a demonstration of the potential savings that a good multiple-copy file allocation could effect. We hope that this will stimulate the reader's desire for an operational distributed data base system. A more comprehensive discussion of the file allocation problem may be found in appendix A.

In general the formulation of a file allocation problem includes two types of transactions: updates and queries. The traffic required for such transactions through a given network depends (among other things) on the number of copies of the files. Since a given update must be seen by (i.e. sent to) all the copies of a file, the more copies we have, the higher the update traffic will be. This traffic will be minimized if there is only one copy of each file. On the other hand, the addition of new copies of a file tends to reduce the network query traffic, up to the point where every site has its own copy of the data base and responds to its queries locally, so that there is no network query traffic. Clearly there is a tradeoff. A simplified example follows:

Let's assume that we have two sites, A and B, that use a given data base system. It costs A the quantity Q_{AA} to process its queries locally and Q_{AB} to process them at B. A price U_{AA} must be paid by A to locally process the updates it generates. A price U_{AB} must be paid to send them to B and process them there. The storage cost to locate the data base located at A is σ_A. In a similar fashion the quantities Q_{BB}, Q_{BA}, U_{BB}, U_{BA} and σ_B are defined.

If there is a single copy of the data base at A, we will have to pay $Q_{AA}+U_{AA}+\sigma_A$ to satisfy A's needs and $Q_{BA}+U_{BA}$ to satisfy B's. If we decide to allocate a second copy to B we would probably save by being able to solve B's access needs locally (i.e., Q_{BB} instead of Q_{BA}) but we would then have to pay for additional storage (σ_B) and transmission of updates ($U_{AB}+U_{BB}$). Thus the cost shifts from $Q_{AA}+U_{AA}+\sigma_A+Q_{BA}+U_{BA}$ to $Q_{AA}+U_{AA}+\sigma_A+Q_{BB}+U_{BB}+\sigma_B+U_{AB}+U_{BA}$ as we move from one copy to two copies.

Comparing the above formulas term by term, we see that both share the $Q_{AA}+U_{AA}+\sigma_A+U_{BA}$ term, but the one-site allocation has an additional Q_{BA} term and the two-site allocation an extra $Q_{BB}+U_{BB}+\sigma_B+U_{AB}$ term. Thus, the two-site allocation is cheaper if $Q_{BB}+U_{BB}+\sigma_B+U_{AB}<Q_{BA}$. It is clear that we could choose these numbers in such a way as to make the single copy allocation more expensive than the multiple copy one. For example, let's assume that the transmission cost is significant, so that it costs ten times as much to process something remotely (i.e., $Q_{BA}=10\ Q_{BB}$). Furthermore, suppose the update activity is so low compared to query activity that update cost represents at most 10% of the query cost (i.e., $Q_{BA}\geq 10\ U_{AB}$, $Q_{BA}\geq 10\ U_{BB}$). Under these assumptions, the additional cost of the two-site allocation ($Q_{BB}+U_{BB}+U_{AB}+\sigma_B$) is less than $0.3\ Q_{BA}+\sigma_B$. Then the two-site allocation is cheaper than the single-site one if

$$0.3\ Q_{BA} + \sigma_B < Q_{BA},$$

or $$\sigma_B < 0.7\ Q_{BA}.$$

The cost of storing a moderate-sized data base should be a fraction of the cost of querying it heavily - especially by way of a network connection - so this inequality is quite likely to be satisfied.

By way of generalizing this example, we introduce Casey's model [C1]. In Casey's model the transmission of queries and updates is as described before and leads to the cost function:

$$C(I) = \sum_{j=1}^{n} \left[\sum_{k \in I} \Psi_j d'_{jk} + \lambda_j \min_{k \in I} d_{jk} \right] + \sum_{k \in I} \sigma_k$$

Where:

I = index set of sites with a copy of the file

n = number of sites in the network

Ψ_j = update load originating at site j

λ_j = query load originating at site j

d_{jk} = cost of communication of one query unit from site j to site k

d'_{jk} = cost of communication of one update unit from site j to site k

σ_k = storage cost of file at site k

$\lambda_j \min_{k \in I} d_{jk}$ = cost of sending queries to closest copy

$\Psi_j d'_{jk}$ = cost of sending update to the k^{th} copy.

Using this cost function to determine the best strategy we can now solve the file allocation problem for any number of sites. For example, let's assume that we have a network with 5-sites, all of which make 24,000 queries per month, each one of them requiring one thousand bits of information to be transmitted. Suppose that each update requires the same amount of information transfer as a query, but that there are only 2, 3, 4, 6 and 8 thousand updates per month for sites 1 to 5 respectively. (Assuming that update units and query units are the same size means that $d_{ij}=d'_{ij}$.) Let the cost of transmitting a megabit be as indicated in table II.2-1. Finally we will assume that memory is abundant and cheap and hence that its cost may be neglected. ($\sigma_k=0$ for all k). We now have a problem similar to Casey's five-node example [C1].

If we neglect node 5 for a moment, we will find that the best strategy (minimum C(I)) is to store a copy of the data base at every node. This will

	To Node				
From node	1	2	3	4	5
1	0	6	12	9	6
2	6	0	6	12	9
3	12	6	0	6	12
4	9	12	6	0	6
5	6	9	12	6	0

Table II.2-1

Transmission cost per megabit in dollars

result in a $384 cost per month as opposed to, for example, $768 per month if we allocate a copy at site 1 only.

If we go back to the 5-node case we will find that the optimal allocation would be to locate copies at nodes 1, 4 and 5. The complete listing of costs for this example is shown in table II.2-2.

A similar example for a 19-node version of the ARPA Network is also presented by Casey in [C1]. In that example multiple copy allocations are again shown to be cheaper than single copy versions in four out of five parameter selections. For example, in the "10% case" (i.e., update load = 10% of query load) the optimum turns out to be allocation 2, 10, 14 with a $117,544 cost (per unit time) while allocation 1 would cost $330,748 and allocation 10 (the best single site allocation) would cost $237,446.

It should be noticed that, in general, Casey does not have an easy and fast method to obtain the optimum. In many cases a search throughout the complete 2^n possibilities (for n nodes) will be needed to obtain the optimum. This exhaustive search is feasible only for relatively small networks and becomes prohibitive very fast (e.g., extrapolation from Casey's data yields an estimate of 136 years to solve a 50 node problem [B4]).

I	C(I)
1	960
2	972
3	1030
4	918
5	915
12	852
13	774
14	726
15	867
23	856
24	730
25	735
34	804
35	729
45	753
123	810
124	762
125	759
134	756
135	753
145	705*
234	760
235	765
245	717
345	711
1234	792
1235	789
1245	741
1345	735
2345	747
12345	771

Table II.2-2

Complete enumeration of all the costs C(I) for the alternative allocations in Casey's 5-node example. The optimum (*) is allocation 1, 4, 5.

Up to now we have been following Casey's ideas, but others are also available. Chu [C5] presents a second alternative. In Chu's model all queries and updates (transactions) cause the data base manager to return a piece of the data base. If the transaction is an update the user sends back the revised information to the data base manager. The cost formula includes storage costs, cost of transmitting the transactions and, in the case of updates, the cost of transmitting the modified data back to the data base manager. The way Chu approaches optimization is drastically different from Casey's method. Chu formulates his problem as a zero-one integer programming problem. The objective cost function C to be minimized is defined as:

$$C = C_{storage} + C_{transmission}$$

where:

$$C_{storage} = \sum_{i,j} C_{ij} L_j X_{ij}$$

$$C_{transmission} = C_{transactions} + C_{modifications}$$

$$C_{transactions} = \sum_{i,j,k} \frac{1}{r_j} C'_{ik} \ell_j u_{ij} X_{kj} (1-X_{ij})$$

$$C_{modifications} = \sum_{i,j,k} C'_{ik} \ell_j u_{ij} X_{kj} P_{ij}$$

X_{ij} = 1 if j^{th} file is stored in the i^{th} computer and 0 otherwise; i.e. the X_{ij}'s are the zero-one variables

L_j = length of j^{th} file

C_{ij} = storage cost per unit length of the j^{th} file at the i^{th} computer

r_j = number of copies of the j^{th} file (which may be computed from availability constraints; see the discussion of r in section II.3.2)

C'_{ik} = transmission cost from the k^{th} computer to the i^{th} computer per unit file length

ℓ_j = length of each transaction for the j^{th} file (i.e., amount of data to be shipped)

u_{ij} = the request rate for the entire or part of the j^{th} file by the i^{th} site per unit time

P_{ij} = the fraction of the time that a transaction for the j^{th} file by the i^{th} site is a modification (i.e., that the data must be shipped back)

This formulation follows a series of arguments and assumptions that we won't discuss here. Chu's approach leads to a non-linear zero-one integer program. However, he demonstrates how this problem can be linearized at the expense of increasing the number of variables. Chu's approach is very sound mathematically, but in general, as Levin [L2] points out, the solution of a moderately sized network problem is computationally very expensive.

A third alternative to Chu's and Casey's methods is presented by Levin [L2]. Levin's work is closely related to Casey's but with a few major distinctions. He considers the allocation of programs as part of the optimization mechanism. He also considers situations in which query and update patterns are not known a priori and an adaptive system is required.

Our main intention in the latter part of this section has been to demonstrate that a multiple copy allocation can in fact be cheaper than the single copy alternative. However this evaluation is dependent on the way we look at the file allocation problem, i.e. the model used. Three models were discussed to show the flexibility available. More details on file allocation are presented in appendix A and in [B4].

II.3 Availability

II.3.1 Introduction

Chu [C5] and Belford et al. [B2] have studied the effect that multiple copies of files in a distributed data base system have upon the availability of a given file. Belford's results are more recent and much more complete than

Chu's. Because of the completeness of reference [B2], our discussion below will follow the treatment there fairly closely.

The term availability will be used "to mean the fraction of time that a data base is available to respond to user requests or queries" [B2].

II.3.2 Chu's work [C5]

Chu presented his study embedded in the development of his one-zero integer programming approach to the file allocation problem. Chu's intention was to obtain a formulation that could allow him to include availability as one of the constraints of his integer programming problem. He assumes that:

1. all computers in the network have an availability of A = up-time/total-time = up-time/(up-time + down-time),
2. the network is completely connected (i.e. there exists a direct communication channel between each pair of computers), and
3. all communication channels have an availability of C (defined similarly to A).

Under these assumptions, Chu finds that the availability of a given file with r copies (at different sites) would be

$$A(1 - (1 - AC)^r). \qquad \text{(II.3-i)}$$

Thus, the availability constraint could be included as a constraint $A(1 - (1 - AC)^r) \geq a$ for a given a. The value of r could easily be precomputed and presented to the integer programming problem as a lower bound on the number of copies for the given file.

The formula II.3-i can be obtained by noticing that:

AC — is the availability of a computer and the channel that leads to it; i.e., the probability that we can establish a direct connection to a given site.

(1-AC) — is the probability that the site is not available to us.

$(1-AC)^r$ represents the probability that r sites are not available to us.

$1-(1-AC)^r$ denotes the probability that a least one of r sites is accessible, and finally

$A(1-(1-AC)^r)$ indicates the probability that a given user who uses a computer with availability A (and does not have a local copy) is able to access at least one of a given set of r computers.

As an example of the utilization of the expression II.3-i, let's assume that we have a network of computers with a down-time of one hour per day (A = 23/24 = .9583) and channels with a .99 availability. Expression II.3-i will tell us that the worst-case availability (no local copy) is .9092 for r=1, .9558 for r=2 and .9583 for r=4. This implies that if we have a single copy we should expect about 65.4 hours of down-time per month, about 31.8 hours for 2 copies and 30.0 hours for 4 copies. These down-times are clearly bounded by A; i.e., by 30 hours per month. It is not hard to imagine a context in which an improvement from 65.4 to 31.8 hours of down time per month would be highly desired.

Some of the problems with Chu's approach are:

a. Chu seems to assume in II.3-i that the user's site does not have a local copy. No explanation is given for this assumption. If the user is allowed to have a local copy his availability will be A. Thus expression II.3-i could be interpreted as the lower bound for the availability, since $A > A(1-(1-AC)^r)$. This seems to be a valid interpretation. If we are interested in average availability we could transform the expression II.3-i to

$$A(1-p)(1-(1-AC)^r) + pA \qquad \text{II.3-ii}$$

where p is the proportion of the users that have a local copy. Since p is dependent on r, it might be difficult to determine an r from a given value for the expression II.3-ii.

b. Chu's approach implicitly assumes that the files are static or, if dynamic, that they are updated simultaneously. In this circumstance, when a site fails we can simply switch to another one and continue our work. This is hardly the real case, and an additional recovery time might be required.

c. The assumed topology (complete connection) and routing technique (direct communication) are the easiest alternatives available. These assumptions, as well as that of the uniform behavior of communication channels and computers, give rise to the simple parameters A and C. In real life this might be an oversimplification. The study of the multiple elements hidden behind A and C (routing, data communication processors, etc.) is an interesting topic.

An extension of Chu's model may be found in [B2, appendix 1]. Different values for the availability of different computers are allowed, and only slightly more complicated expressions are obtained.

II.3.3 The work of Belford et al. [B2]

The approach of Belford and co-workers is much more realistic than Chu's. Rather than assuming that files are static or that updates are applied simultaneously, they assume that there exists a single master copy and all other copies are treated as backups. All updates are applied to the master copy as soon as possible. Updates are applied to the backups according to one of the following methods:

1. Running spares. Backup sites apply the updates almost as fast as the master, and
2. Remote journaling. Backup sites periodically receive up-to-date copies of the master copy. Between these times, updates are only

journalized at the backup site (i.e., only the master copy processes the updates).

Some of the results of the study indicate that, from the availability point of view, the usefulness of method 2 is a function of how long it would take to bring the backup copy up to date. If the backup is stored on tape then availability would only be slightly improved, and the additional work would hardly be justifiable on the basis of the availability improvement. (Actually this kind of backup is used to prevent data base destruction.) This is simply because by the time we get the backup copy in a usable state the primary copy is likely to be up again.

It is important to note that remote journaling over a network is not feasible on an extended basis. Moderate-sized files would take hours to transmit through the type of communication lines currently used. For example, in the ARPA Network it would take more than 1 1/2 hours to transmit a file of 10^7 bytes.

With availability in mind, we thus turn to method 1. There are various possible strategies as to what to do when the primary fails. If we allow a backup copy to take over the primary's role only while the primary is down, we end up with what Belford and co-authors present as strategy 2. For this strategy the availability is expressed by:

$$A_2 = 1 - \frac{D+L+kY}{F+X+kX}$$

where:

F = mean time between computer failures, assumed to be the same for all host computers.

X = expected time to repair computer.

L = expected time to load the data base copy at the remote site.

Y = time that the audit trail of updates has been growing (i.e., time since the copy was correct).

k = the ratio of update arrival rate to update processing rate.

D = time delay between when the master fails and when the remote site determines this fact and starts to get its copy ready for use.

Using the same notation, we have that the single copy availability should be

$$A_o = \frac{F}{F+X+kX}$$

If we study the improvement (I) that we get by using A_2 over A_o (i.e. $I = (A_2-A_o)/A_o$) we have that:

$$I = \frac{X+kX-D-L-kY}{F}$$

Assuming some reasonable parameters (k = .01, D = .01 hrs., X = 1 hr., L = .5 hr. and Y = F) we obtain figure II.3-a (figure 3 in [B2]). In this figure we have plotted I as a function of F (i.e. $I = \frac{.5}{F} - .01$). The effect on A_o of varying F has been plotted as a reference.

One should not be misled by the small relative values of I. For example, for F = 20 the improvement is .015. This implies that the down-time would change from 35 to about 24 hrs/month if we choose to have more than one copy.

In general the work of Belford and co-workers is much more comprehensive than Chu's but it is still not a complete solution. Some of the missing topics are:

1. Inclusion of the communication channel in the formulations. The decision to ignore communication channels is probably based on studies of the ARPA Network [F1]. More than one channel failure is required to block the transmission of a message through the ARPA Network,

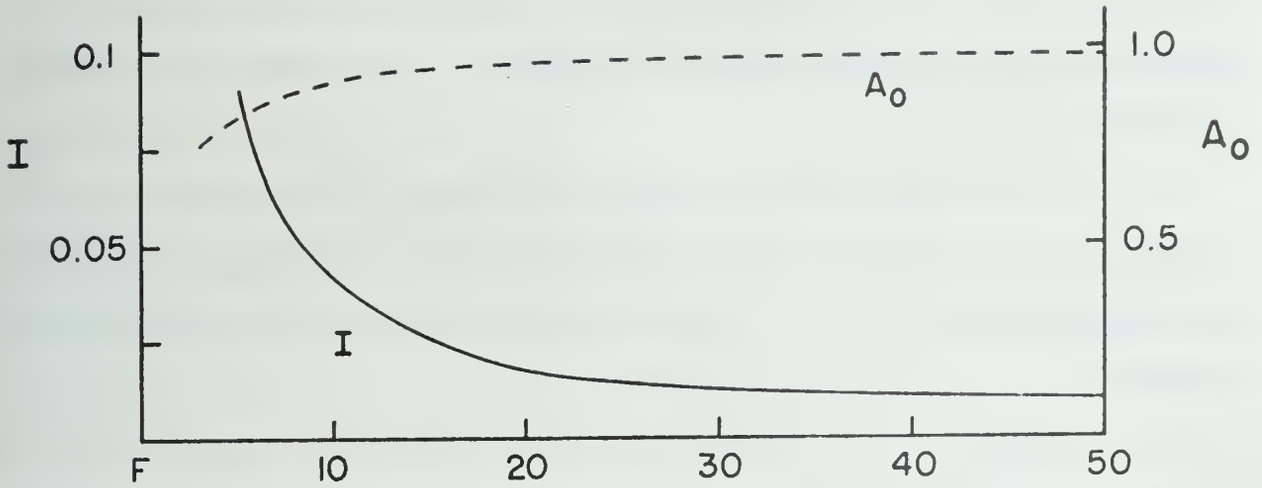


Figure II.3-a

Single-site availability A_o and fractional improvement I through use of strategy 2. Parameters are k = 0.01, D = 0.01 hr., X = 1 hr., L = 0.5 hr., and Y = F.

making the availability of Chu's communication channel essentially equal to one. It could be useful to show convincingly that this logical omission is actually correct.

2. Communication processors. ARPANET experience indicates that the time between failures of network communication processors (IMP's) is much longer than that of the computers (hosts) they serve. Thus, it is reasonable to neglect the communication processors in an ARPANET-like environment, but probably not in general.

Due to these two omissions, everything related to network topology, routing and connectivity was ignored.

II.3.4 Discussion

Our main interest in availability is to prove that it is improved by a distributed data base management. Both studies in the literature give us the

necessary tools to measure availability for the models assumed. Armed with these tools, we could easily compute the number of copies that our distributed data base system should have.

Both studies neglect to consider (and we will do so too) the inverse problem. Given a certain number of copies, what should be the best allocation to maximize availability? The answer to this question, although interesting, is tangential to our study and won't be considered.

What we were looking for was to study the impact of multiple copies on data base availability. The discussion and examples presented clearly justify a distributed data base system when availability is at stake.

II.4 Response time

Like availability, response time might be given as an *a priori* design restriction. It might be one of the factors taken into account in a formulation of the optimal file allocation problem or it might be studied as an independent topic.

Chu [C5] has adopted the first approach. In the formulation of his integer programming solution for the file allocation problem, he introduces a set of restrictions:

$$X_{kj} a_{ijk} \leq T_{ij} \text{ for } i \neq k,\ 1 \leq j \leq m,$$

where:

X_{kj} is 1 if and only if the j^{th} file is stored in the k^{th} computer, and is 0 otherwise,

a_{ijk} is the expected time for the i^{th} computer to retrieve the j^{th} file from the k^{th} computer, and

T_{ij} is the maximum allowable retrieval time of the j^{th} file by the i^{th} computer.

(This simple formula requires the assumption that there is only one copy of each file in the system.) Chu uses a simple model and queueing theory to obtain an approximation to a_{ijk}.

We can treat response time as the objective function in the file allocation optimization. For example, for a given availability (i.e. for r copies of the file), we could find the allocation that minimizes response time. There are no reports of this approach in the literature.

There is only one contribution to the generalized study of response time in a distributed data base system. Response time can clearly be improved by having as many copies of the data base as possible. It can also be improved by locating these copies at appropriate sites. Finally, it can be improved by load sharing. Belford et al. [B1] developed a mathematical model for response time in a distributed data base environment. The concern was to establish when response time can be improved by sharing parts of the load of a given site. If the local load satisfies a given formula, sharing is initiated. For simplicity, it was assumed that all queries arise at a single site. Different cases are considered, such as uniform or non-uniform distribution of excess queries to the remote sites.

Distributed data bases will unquestionably improve response time. This improvement can be produced by proper choice of file allocation, load sharing, or simply by having a multiplicity of copies of the data base.

Chapter III. EXISTING MODELS OF DISTRIBUTED DATA BASE SYSTEMS

III.1 Introduction

Once we are convinced (by the previous chapter) that implementing a distributed data base system is a sensible thing to do, we have to decide how to do it. Casey's work [C1] assumes that queries and updates should be handled as described in figure III.1-a. Queries are sent to the closest site that possesses a copy of the data base and updates to every site that maintains a copy. ("Closest" may be defined broadly in terms of lowest cost.) Unfortunately this simple approach does not work in general. The reason is that we have problems in synchronizing our various copies of the data base. The synchronization problem will be best explained by the presentation of the following three examples.

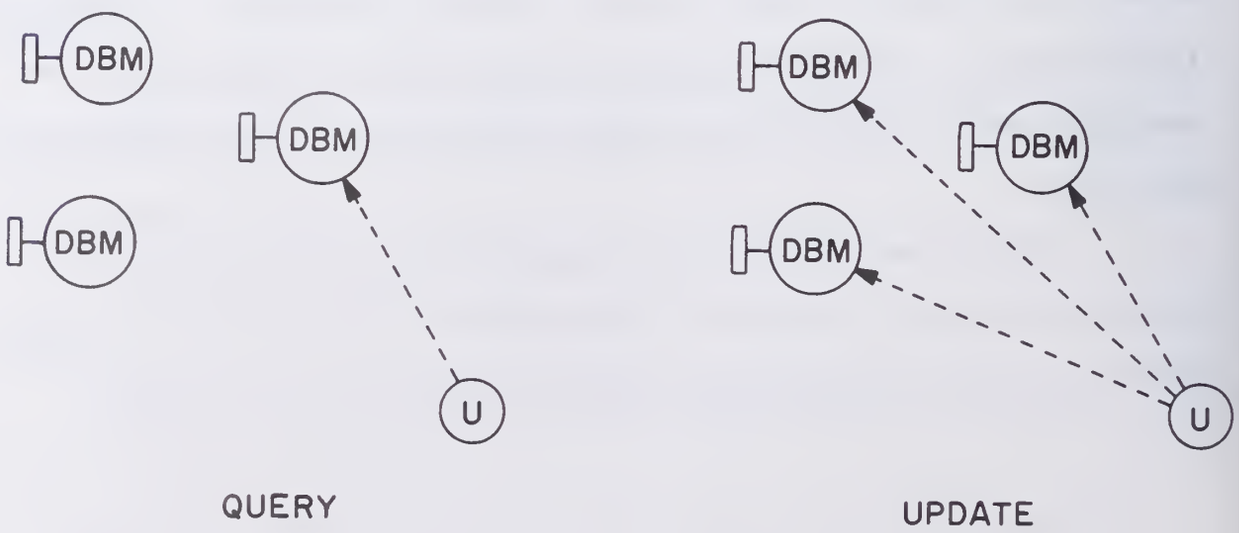


Figure III.1-a

Illustration of Casey's model for treatment of queries and updates. U is the user site while DBM is a data base manager with a local copy under its supervision.

EXAMPLE 1. No Problem. We look at two sites, A and B, maintaining copies of the data base (see figure III.1-b). Two additional sites (they could also be A and B) generate an update each. The updates are nearly simultaneous and both affect the same field x. The first one adds 1 to x, the second one adds 2. Because of the network topology and delays, it turns out that the updates arrive at B precisely in the opposite order that they follow for A; i.e., A first adds 1 to x and then 2 more; B first adds 2 to x and then 1 more. No problem is evident in this example.

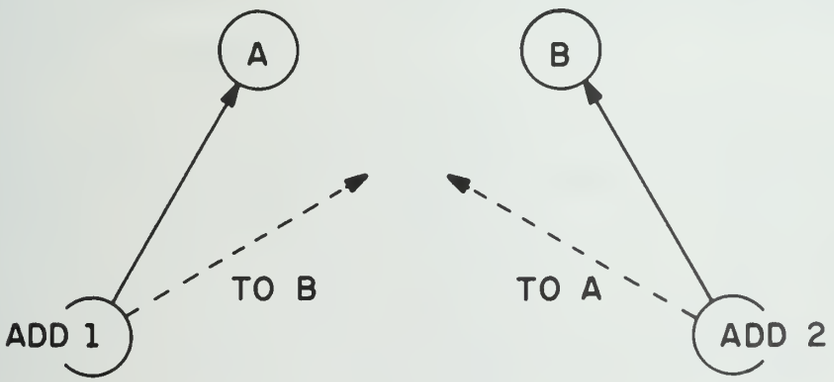


Figure III.1-b

Graphical explanation of what occurs in example 1. The dashed lines indicate later arrival due to delays.

EXAMPLE 2. Numeric Inconsistency. We now change the updates to read "increase x by 10%" and "add 10 to x" (see figure III.1-c), leaving everything else the same. The results are that A and B won't agree any more. The value for x at site A will be 1.1 x +10 while for B it will be 1.1 x +11 (= 1.1(x+10)).

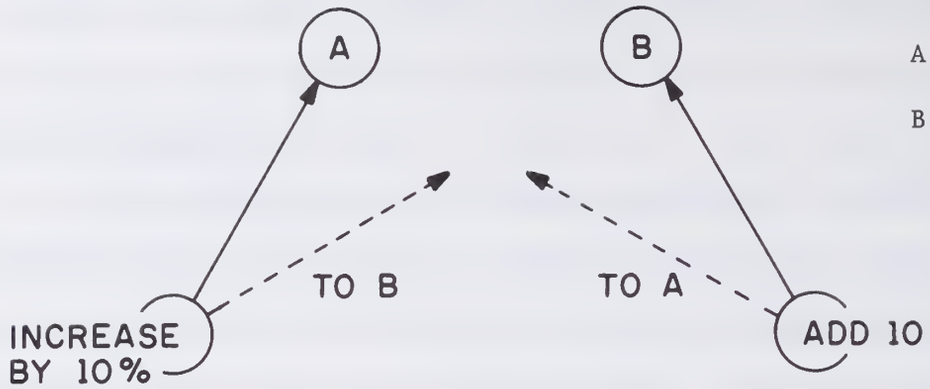


Figure III.1-c

Graphical explanation of example 2. The dashed lines indicate later arrival due to delays.

EXAMPLE 3. Logic Inconsistency. If we change the updates once again to "change all entries with a w in field x to z" and "change all entries with a w in field x to y" (see figure III.1-d), we get involved in a logic inconsistency. Site A, upon arrival of the first update, will change field x to z and won't be able to recognize or service the second update (change field x to y). A similar problem will arise at site B and cause the undesired consequence: logic inconsistency.

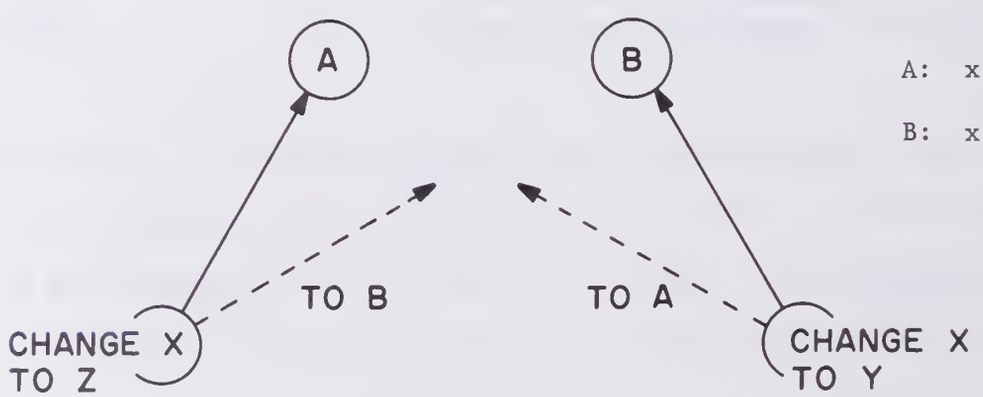


Figure III.1-d

Graphic description of example 3. The dashed lines indicate later arrival due to delays.

We therefore must discard Casey's approach, at least in its primitive state, and start looking for some alternatives. In the present chapter we will present three models formulated with this purpose in mind. These models are:

a) Johnson's model (section III.2),

b) Bunch's model (section III.3), and

c) the Reservation Center model (section III.4).

The relative advantages and disadvantages will be the subject of the next chapter.

III.2 Johnson's model

III.2.1 Context

Johnson's model is described in two papers [J1] and [J2]. This model postulates the existence of data base managers as interfaces between users and data bases. Each data base manager maintains a local copy of the full data base. Johnson's data base consists of a collection of entries (or records). Each entry is defined as a five-tuple (ℓ,V,F,CT,T) where:

ℓ is a selector (or location),

V is the associated value,

F is a deletion flag (see section III.2.3),

CT is the creation time-stamp (see section III.2.3), and

T is the time-stamp of the last operation which modified the entry (see section III.2.2).

Thus each data base entry contains only one data item (V) of unspecified characteristics. The selector is used to identify the item; only one item is identified by each selector. We can think of the selector as the address of the data item.

Updates are transmitted from one data base manager to another by sending a new data base entry. Thus updates and data entries have the same format.

III.2.2 Synchronization mechanism

Inter-manager synchronization is obtained by using the time-stamp component (T) in the updates. A time-stamp is defined as a pair (τ,D) where τ is the time and D is the identification of the site where the entry or update originated. It is thus possible to establish a unique order of precedence for the 5-tuples. A time-stamp $T1(\tau_1,D_1)$ is said to precede (or "be older than", or "be smaller than") another time-stamp $T2(\tau_2,D_2)$ if and only if τ_1 is smaller than τ_2; or, if $\tau_1=\tau_2$, if D_1 is smaller than D_2. A 5-tuple precedes another 5-tuple if its time-stamps do so. Two 5-tuples with the same time-stamp are considered to be the same 5-tuple. Thus only one update per time-instant is allowed.

For example, let us assume that we have two updates, u_1 and u_2, generated within a short interval of time. Update u_1 originates at site D_1 at time τ_1 (say 10:00) and is designed to modify location ℓ to value V_1. Update u_2 enters the system at site D_2 when the clock indicates τ_2 (say 10:01) and is intended to assign value V_2 to the same location ℓ. Data base managers A and B receive these updates in a different order (see figure III.2-a). If each data base manager applies the updates in the order it gets them, we end up with an inconsistency. That is, the data base at site A ends up with V_2 at ℓ while site B has V_1. However, the time-stamps could be used to detect this situation. When u_1 arrives at site B, the data base manager could determine that u_1 is older than u_2 and discard it. Both data base managers would then end up with the same value (V_2) for location ℓ.

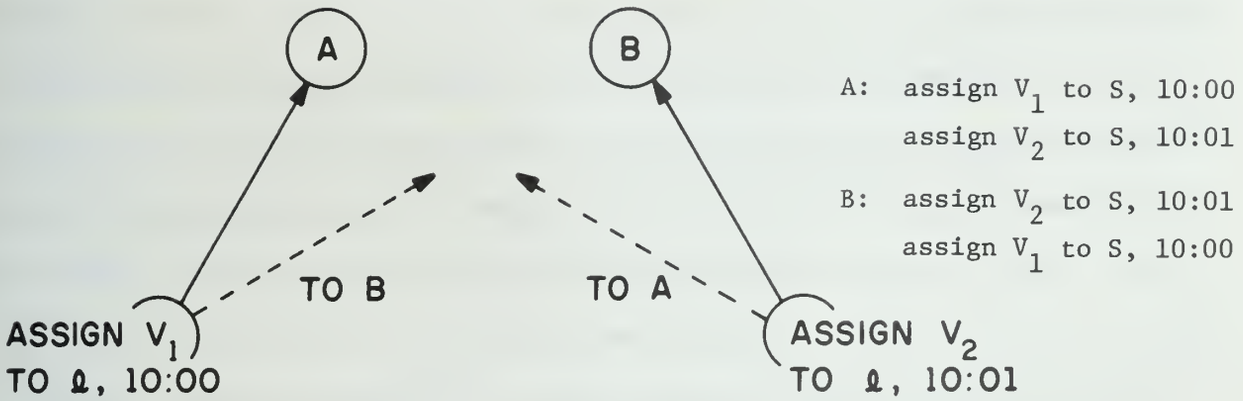


Figure III.2-a

Update transmission for example in text.
Dashed lines indicate later arrival.

III.2.3 Operations available

Four operations are defined:

a) Selection. Given a selector, the current associated value V is returned.

b) Assignment. A selector and value are given and the given value replaces the old value associated with the selector.

c) Creation. A new selector and an initial value are given and a new entry is added to the data base.

d) Deletion. A selector is given and the existing data base entry indicated by the selector is removed from the data base.

We next discuss the problem of incorporating these operations into the data base scheme described above. As we shall see, there is considerable variation in the difficulty of handling these operations.

Selections correspond to queries in the terminology that we have been using. A selection transmits a selector (ℓ) to one of the data base managers,

which in turn responds by sending the associated value (V) in the 5-tuple indicated by ℓ.

Assignments are the basic type of update considered in this model. They are expected to be transmitted to every data base manager. An assignment is designed to replace the old data base entry for the same selector by a new 5-tuple. The new 5-tuple differs from the old one in the V and T elements. V indicates the new value that should be associated with the given selector and T indicates the time-stamp of the update (i.e., the time it occurred and the manager which created the update).

When an update arrives, the manager will decide which of the two 5-tuples for the same selector should remain in the data base (i.e., the one that is already there or the incoming update). This decision will be based upon the time-stamps. The one with a more recent time-stamp will become the new data base entry. If it's the 5-tuple that was already there we simply ignore the incoming update, otherwise the update's 5-tuple becomes the data base entry and is properly stored. For example, in our last example data base manager A will have the 5-tuple $(\ell, V_1, F_1, CT_1, (10{:}00, D_1))$ stored in the data base when the 5-tuple $(\ell, V_2, F_2, CT_2, (10{:}01, D_2))$ arrives. Data base manager A will then store the later 5-tuple in its data base. On the other hand, data base manager B will have 5-tuple $(\ell, V_2, F_2, CT_2, (10{:}01, D_2))$ stored in its data base, and will ignore the u_1 update because it is too old. (We have thus obtained inter-data base consistency.)

Creation can in general be handled in two ways: explicitly or implicitly. Explicit creation requires that a specific create message be provided in order for a data base manager to create a new record. A problem with such a method is that, due to network characteristics, a given site could legally get updates for locally nonexistent entries. (For example, A creates x and notifies

B. B creates x and updates it. C gets B's update before A's creation.) Implicit creation means that if an assignment for an unknown entry is received, the given entry will be created. Thus creations are transmitted as assignments. Implicit creations avoid that problem described for explicit creation. However, a new problem appears: transmission or other errors could produce undesired creations. Johnson's model uses implicit creation.

In this model creations are treated as assignments with one small difference. We delayed until now the discussion of the CT element in the 5-tuples. The CT element corresponds to the creation time-stamp and has the same format of any ordinary time-stamp. A creation entry looks like an assignment; the initial value is stored in V. In addition to this, the time stamp T is duplicated into CT. In an update, CT must contain the time when the entry was created. Therefore the data base manager can distinguish between creations and updates by whether or not T=CT. (The reasoning behind the CT element will become clearer when we discuss the delete operation.) Functionally, creations are treated as assignments; i.e., there must be the same conflict solving based upon T, etc.

The most troublesome of the operations is the deletion. Deletion cannot be allowed to produce the desired effect of immediate removal from the data base, because pending updates (with older time-stamps) could incorrectly cause the entry to be "re-created". Furthermore, it is not possible to simply ignore all future references to the same selector, because a valid re-creation could occur. (One way to avoid this problem is not to allow entries to be re-created. But this is not assumed in this model.) To solve this problem, the F and CT elements were introduced into the 5-tuple (ℓ,V,F,CT,T), where F is a deletion flag and CT is the time-stamp of the creation, as mentioned before. A creation will have CT=T and F = not deleted, an assignment CT<T and F = not deleted, and

a deletion F = deleted. Under this structure an update is applied only if its creation time-stamp is no smaller than the creation tag of the data base entry, and, in case of equality, only if the data base entry's F field indicates "not deleted."

Upon receiving a deletion, the data base manager simply stores it as it would an assignment, replacing the old entry by the new one with the "deleted" flag. As mentioned before, data base entries are not immediately removed nor the space released. This approach is taken since we have to keep the deleted entry to be able to detect (and ignore) all pending updates (with the same creation time-stamp) for that record. If we had some other means of detecting this situation we could safely remove the deleted entry. Johnson's model provides the following detection mechanism. If we require that every data base manager deliver its own modifications in a strict order, then each data base manager can maintain a vector of length n (number of data base managers) in which the j^{th} component is the latest time-stamp received from the j^{th} data base manager. These vectors could continuously be exchanged between the data base managers, giving each of them a matrix A (nxm) in which the (i,j) entry will indicate the last time-stamp known to have been received by data base manager i from manager j.

Now we are in the position of being able to physically remove entries, because we have enough information to establish whether there are updates that could affect the deleted entry. Requiring that all entries in the A matrix be greater than the time-stamp of an entry to be deleted is more than enough to guarantee safe removal. Actually it's enough to ask less. Johnson and Beeler mention [J1, p.7] that a data base manager could remove a deleted entry whose deletion originated at site k if all entries in the k^{th} column of the A matrix

are larger than or equal to the time-stamp of the deleted entry. This implies that all data base managers have received the delete message.

A refinement to the A-matrix method is also presented. The data base managers exchange only the oldest entries from the vectors of latest update time-stamps received from other mangers. This reduces the required memory space.

In general we question the scheme for physical removal of a deleted entry presented in this model. We will defend our position in section IV.2.

III.2.4 Miscellanea

Johnson and co-workers do not give any detail about any action taken to detect lost messages. The dependence of the A matrix upon the correct (in the sequential sense) transmission of updates makes it important to establish a mechanism capable of such action. A complete journalizing (to keep updates for failed managers) and acknowledgment scheme would be required in general. No specific difficulty that could justify further discussion of this matter is expected.

III.3 Bunch's model

III.3.1 Context

Bunch presents his model in [B5]. His interface between user and data base is divided into two parts: The user-controller and the data base manager. Each data base manager has a data base under its supervision. Each user-controller is responsible for interfacing users and data base managers.

In this model there are no restrictions upon the organization of the data base. Any kind of organization and/or addressing scheme could be handled.

III.3.2 Synchronization mechanism

This model separates the data base managers into two categories: One of the data base managers is the primary; all the others are known as the backups. All the updates are initially sent to the primary. The primary is then in charge of broadcasting them to all the backups.

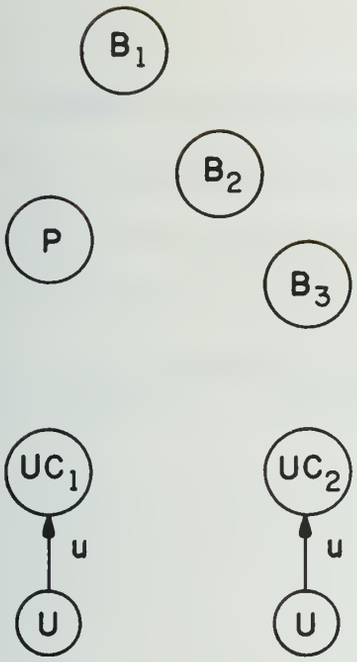
When a user wants to make an update, u, he will submit it to the user-controller. The user-controller will attach a name or tag[1] (s) to the update and send it to the primary. The primary will assign a universal tag (U) to each update and substitute it for the s tag. The pair (u,U) is then broadcasted to all the other data base managers (i.e., to the backups). The U tag will be sent to the user-controller for proper cross reference as will be discussed later.

The universal tag U is expected to follow a precise ordering. Based upon this tag every backup manager should be able to detect any missing updates. Application of the updates at each data base manager should follow precisely the same order, as indicated by U. If a backup detects that an update is missing, he is forbidden to apply any other update until the missing update is recovered and applied in the correct (and unique) order as indicated by the universal tags. For example, suppose we have two users that want to store values V_1 and V_2 (respectively) at the same location ℓ. The sequence of events described in figure III.3-a represents a possible outcome of this situation.

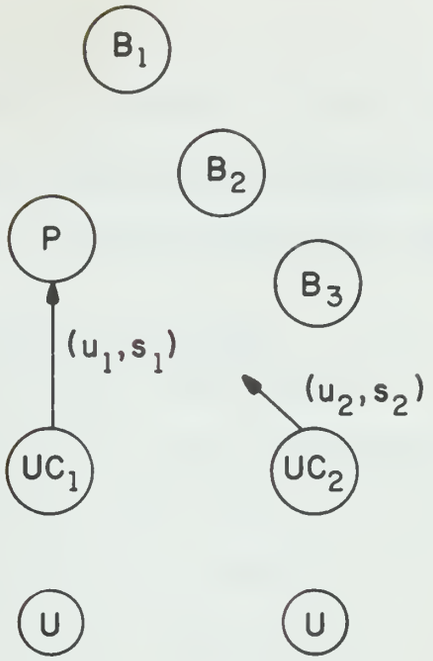
From the sequence of events described in figure III.3-a, we see that every data base manager is expected to get the updates in an order consistent with the universal tags associated with them. If this does not happen (U_2 arrives before U_1), an appropriate procedure could be triggered to recuperate from this fault. Thus all copies of the data base should automatically end up

---

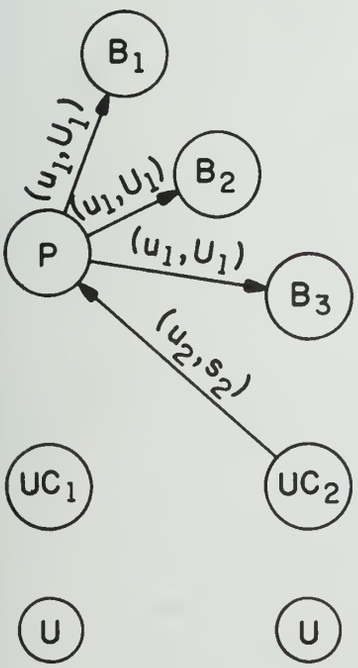
[1] The s tag is intended to guarantee the complete communication between controller and primary; i.e., to detect missing updates.



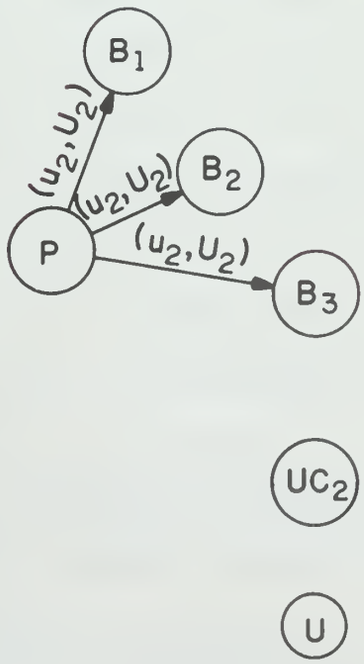
i) User-controllers UC_1 and UC_2 get the updates u_1 and u_2 (respectively) from associated users.



ii) controller UC_1 gets the update to the primary first.



iii) Primary broadcasts u_1 to all backups after attaching the universal tag U_1. Update u_2 arrives at the primary.



iv) Primary broadcasts update u_2 and corresponding universal tag U_2.

Figure III.3-a

Sequence of events to perform two almost simultaneous updates u_1 and u_2 in Bunch's model.

with value V_2 at location ℓ. This contrasts with the time-stamp scheme, in which a data base manager has no way of knowing about a delayed, out-of-order update until after it arrives.

III.3.3 Operations available

Bunch's model supports three types of operations: a) critical read, b) non-critical read, and c) modifications.

a) Critical reads are intended to obtain the most recent information. Upon reception of a request for a critical read, the user-controller will transmit it to the primary, which controls the most up-to-date copy. The primary's reply will be considered as the satisfaction of the critical read. This operation is clearly a special type of query.

b) Non-critical reads do not require the latest information. The user-controller is thus able to use a different criterion to decide which one of the data base managers could be asked to reply. Location, queueing delays, costs, etc. could assist in making this choice. This is the classical query. Only this type of query is supported in the other models.

c) Modifications. In this category we have any operation that can cause the response to a subsequent critical read to change; i.e., modifications are what we have been calling updates (e.g., assignments, creation, deletions, increments, relational operations, etc.).

The explanation of Bunch's treatment of the previous example (see figure III.3-a) demonstrates the way updates are carried out.

III.3.4 Miscellanea

Bunch does not introduce a protocol to deal with detection and correction of lost messages. However, we do not envision any complications arising from the use of s and U for such purposes.

Computer and communication line failures are a matter of concern in this model. If a backup copy fails, then the primary has to take proper notice and action. The primary is expected to stop any broadcasting to the failed computer, while at the same time making sure that all new updates are journalized. The way this is taken care of is by letting each user-controller be in charge of journalizing the updates it generates. The primary sets a counter to be able to detect when all backup copies have acknowledged the reception of the update; at that moment it will notify the user-controller. The user-controller can then go ahead and eliminate the update from its journal. When a backup failure is detected, the primary will notify all other backups of the failure. All working backups are then expected to start journalizing the updates they get from the primary. When the failed backup comes up, the primary will designate one of the data base copy holders (perhaps itself) to update the recovering backup from its journal. When no backups are down, the primary notifies all backups that they may stop journalizing.

When the primary copy fails things are much more complicated. First of all a new primary must be elected. Secondly the status of the failed primary must be figured out by the newly elected primary.

To elect a new primary a successor list is provided by the primary. When a backup notices the failure of the primary, it will notify the first backup in the successor list. When a backup gets such a notification, it will send a message to all other backups to ask for its acceptance as the new primary. This message includes the time at which the successor list (the authority of

which they are using to claim the primary post) was generated. All backups receiving this message will respond either "yes" or "no" according to their acceptance or denial of the precedence of the sending backup. Only one of the backups will get only "yes" answers. It will then declare itself the new primary and start recovery procedures.

To become operationally the primary, the newly elected primary must obtain a reasonable image of the status when the previous primary failed. To do so it notifies all user-controllers that recovery is in progress. Each controller will send to the new primary all updates that have not been completed (i.e., no notification that every copy has gotten the update has been received). This retransmission of an update will differ from the initial transmission because it will now include the corresponding universal tag U, which was sent by the previous primary as an acknowledgment of the reception of the update. The new primary analyzes these updates, applies to its local copy all the ones it hasn't seen, and schedules for future processing the remaining ones that the previous primary did not service (i.e., no U tag is known by the user-controller). After all controllers have answered, operations are re-initiated.

Bunch neglects to mention specifically how the other managers will get the updates that they are missing. However, this seems unnecessary because the protocol that takes care of missing messages (by verifying correct U sequence) could easily take care of this situation.

III.4 The Reservation Center model

III.4.1 Context

Two obvious problems with the previous two models guided us to the development of a third model. In Johnson's model the clock synchronization could be critical. If a clock for a given data base manager is set forward, it will actually block the application of updates to any record touched by this

data base manager until the moment when the other managers' clocks can overcome the setting difference.

On the other hand, Bunch's model is completely centralized. Its performance and throughput highly depend on the performance of the primary; i.e., we have a potential bottleneck.

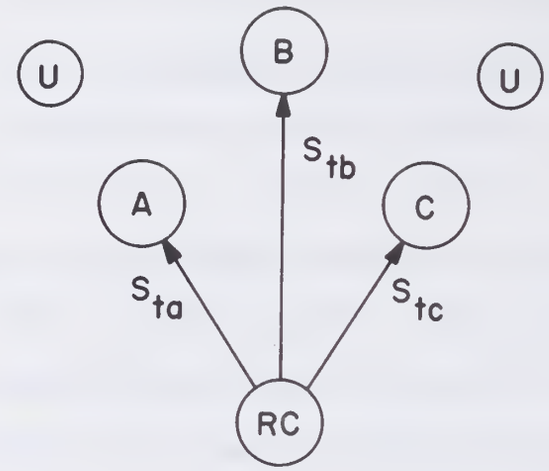
The previous two models represent two extremes, complete decentralization (Johnson) and complete centralization (Bunch). The third model - the Reservation Center model - is intended as a middle way between these extremes. The other context items (data base managers, data base entries, etc.) are considered to be treated similarly to Johnson's model.

III.4.2 Synchronization mechanism

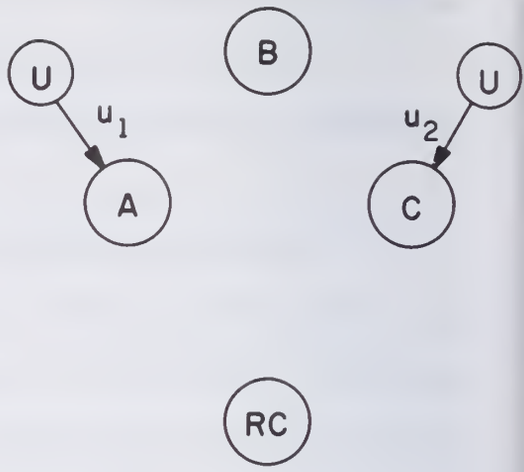
The model is built around a reservation center facility which is responsible for issuing sets of tickets (s_{tj}) to all data base managers. (See figure III.4-a.) A set of tickets is any combination of numbers or instructions capable of generating a collection of ordered names or numbers. For simplicity we will assume that a set of tickets is a collection of numbers, and we will carefully note when this assumption is violated.

When a data base manager gets an update, it will obtain a ticket out of its set of tickets and attach it to the update. This ticket number will be instrumental in establishing the order which the updates should follow. A higher ticket number is considered more recent than a lower one and the higher will override the lower if a conflict occurs.

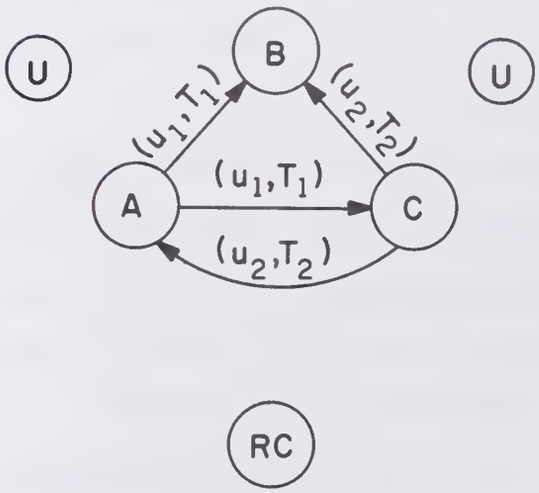
The reservation center generates and transmits new sets of tickets every m seconds, where m is a design parameter. We denote the set of tickets generated at the i^{th} m-period for the data base manager j as s_{ij}. The only



i) The reservation center (RC) issues sets of tickets to all data base managers. For the purpose of the example to follow, suppose that these include ticket number T_1 for manager A and number T_2 for manager C.



ii) Managers A and C get updates u_1 and u_2 respectively, to which they attach tickets T_1 and T_2 correspondingly.



iii) Managers A and C broadcast their updates. When conflict occurs ticket numbers T_1 and T_2 will be used to assure uniform applicatio

Figure III.4-a

Example of treatment of two almost simultaneous updates (u_1 and u_2) under the Reservation Center model.

restriction imposed by the model on the generation of tickets is that any ticket of s_{ij} be smaller than any one from $s_{i+1,k}$, for all j and k.

In any given instance, the assignment of a sequence number from the current set is left to the application. Clearly this implies that within one collection of sets of tickets (one m-period) there exists an uncertainty with respect to the order the updates are going to follow. We argue that a similar uncertainty is produced by the asynchrony of clock settings (in Johnson's model) or the different network delays (in Bunch's model). The only difference is that this uncertainty has been explicitly included in our model.

One of the features of this model is that it includes Johnson's model as a limiting case. We simply set $m=\infty$ and make the set of tickets a set of instructions to copy the time. If m is small the Reservation Center model seems rather like Bunch's model, but the different approaches to broadcasting the updates (centralized vs. decentralized) make this comparison a little harder to make.

To avoid periods of inactivity due to the exhaustion of the tickets in a given set, we allow each data base manager to keep n sets of tickets. Thus our model could be denoted as RC(m,n) to emphasize the dependency on those parameters. For simplicity, we will continue our discussion for an RC(m,2) model.

Thus each data base manager keeps two sets of tickets, s_1 and s_2. The set s_1 is known as the current set, s_2 as the next set. Whenever a new set of tickets arrives, s_2 becomes s_1 and the incoming set becomes s_2. Whenever the data base manager exhausts s_1 before a new set of tickets arrives, it transforms s_2 into s_1 and notifies the reservation center. The reservation center is then expected to expedite the generation of the next set of tickets, while at the

same time increasing the allocation of tickets to the given data base manager to help to avoid future shortages.

When a new set of tickets (s) arrives, the local time at arrival is attached. The validity of a given set of tickets extends for $2m+\delta$ (in general $nm+\delta$) where m is the expected time until s becomes s_1, (i.e., begins to be used), m is the expected time for s to be used as s_1, and δ is a small tolerance factor for unexpected delays. When this period of validity expires, s_2 is transformed into s_1. An "alert" state could then be entered. During this alert state, the reservation center is watched carefully to determine whether it has failed. A timeout could be used to detect such a failure. Recovery procedures could then be initated. (See section III.4.4.)

III.4.3 Operations available

If we desire to include Johnson's model as a special case of the RC(m,n) model, we have to pay the price of limiting the number of operations. Thus we will assume that the discussion presented in section II.2.3 is applicable here with almost no exception. For reasons presented later (section IV.2), we will slightly modify the procedure for detecting that it is safe to remove a deleted entry.

In the RC(m,n) model a deletion is performed in two steps. 1) When the deletion message arrives, the value V of the corresponding 5-tuple is modified to a special value indicating that the entry has been deleted. 2) Whenever all tickets with a smaller value than the ticket of the deletion message have arrived (this is discovered by examination of the proper row in the A matrix), we can proceed to the physical deletion, if the value V still indicates deletion.

The actual set of operations could be increased if we follow the ideas to be presented in section IV.5. However, section IV.6 will point out the

enormous restrictions on any such extension. For this reason, we won't make an effort to extend Johnson's set of operations. Such an extension would only complicate our comparison of the models (in the next chapter).

III.4.4 Miscellanea

Recovering from a reservation center failure implies first establishing a new reservation center and second providing this new reservation center with a lower bound for the ticket numbers that it should generate. This lower bound should be bigger than the biggest ticket in any of the sets of tickets from the previous reservation center. These two steps could be done at once. When a reservation center failure is suspected, a data base manager will call for an election by sending an election message to all other managers. This message includes its "political promise"; i.e., the lower bound that it would assign if elected. If any of the receiving data base managers thinks that it is more suited for the post of reservation center, it will return a "keep quiet" message and broadcast its political promise. On the other hand the data base manager would indicate its acceptance of a political promise by sending a "submit" message. This process could produce a chain reaction of political promises which should end up with only one manager getting only submit messages. At this moment the given data base manager will set itself up as the new reservation center.

Chapter IV. EVALUATION OF THE MODELS

IV.1 Generalities

In this chapter we will try to make an evaluation of the models described in the previous chapter. From many points of view we are comparing apples and oranges, especially because we are not aiming toward any preestablished environment. With this in mind, we look for features that will characterize the advantages and restrictions of each model.

We will start by analyzing each model separately, (sections IV.2, IV.3 and IV.4), studying completeness and applicability to an environment similar to the one it seems to address. We will then (section IV.5) compare them in an environment which could make use of any of the three models. Next (section IV.6), we will study the limitations that each of the models has if we attempt to apply them in a more general environment. Finally (section IV.7), we will present our conclusions and note the unresolved topics of interest. These topics will be treated in the next chapter.

In our evaluations we will always try to keep in mind that we are looking for a workable version for a distributed data base system. Thus, aspects such as reliability will be high on the list of things to look for in each model. Reliability during normal operation will be considered, and problems encountered will be presented. We will also be concerned with reliability during failures, either single failures (e.g. a single computer goes down) or multiple failures (e.g., several computers fail simultaneously), including network partitioning[1] which is possibly the hardest problem in this area. Some

---

[1] A network is partitioned when it is divided into two or more subnetworks that do not maintain communication between them.

other aspects to evaluate in each model will be the set of available instructions, possible protocol problems, memory requirements, etc.

IV.2 Evaluation of Johnson's model

In studying Johnson's model we first have to correct a small omission in order to make the model workable. Johnson does not mention how and when the vectors of his A matrix should be exchanged by the different data base managers. In his discussion there is no apparent reason why a restriction should or should not be imposed. This seems to indicate that the exchange could be done independently of the update exchange, which is expected to be sequential.

Failing to impose a sequential restriction on the broadcasting of these vectors could produce races, as we will describe below. The A vector from data base manager i could not be sent to data base manager j until all the pending updates from i to j are successfully transmitted. This means that we expect these vector interchanges to follow the same sequential order that the updates do. Thus, if data base manager i wants to send its vector to j it should store it in the proper queue and await the appropriate time.

An example of a race condition follows. Johnson says that a given entry with a delete mark can be physically removed when everybody has been notified of the deletion. Let's assume that data base manager k originates a delete message for record R. A data base manager j will recognize that the delete message has been seen by everybody when the k^{th} column of its A matrix has values larger than or equal to the tag of the update deleting R. However, it could well be that everybody has received the deletion message, but two data base managers, i and j, have been slow in sending each other some older updates for record R. For example, i and j may have received the delete message, but in i's queue for messages to be sent to j there still exists a message referring to R. If manager j deletes the entry because the k^{th} column of its A matrix

satisfies Johnson's condition, then it will incorrectly re-create R when it receives the old update from i.

The particular race condition of the previous paragraph could be avoided if we take one of the following approaches.

a. Instead of checking the k^{th} column, manager j could check the j^{th} row. If all the entries in the j^{th} row are larger than the time-stamp of the deleted entry, the entry can be removed. It is certain that nobody could possibly have an older update for R.

b. Manager j should not re-create an entry from an assignment (CT<T) when it has not seen the corresponding creation (CT=T) and according to the A matrix it should have seen it.

That is suppose the CT indicates that the creation was done by site i. If A(j,i) is greater than the time-stamp of CT then the creation should have been seen. In this case manager j can assume that the entry has been deleted and the assignment should be ignored.

A similar race would occur when a data base manager i, not knowing that a given record R has been removed, generates an assignment for R. In this case our solution (a) is worthless but solution (b) still works.

It is easy to ensure that vector broadcasting follows the same sequential transmission rule obeyed by updates. We will assume this is the case in our discussion.

A possibly serious problem with Johnson's model is a result of the synchronization method used. If it happens that one clock is set forward with respect to the others, the corresponding data base manager will have obtained an inherent priority over all the other data base managers. Its updates will prevail over other updates even though the later ones really occurred more recently. To build a reliable system, this blocking hazard must be avoided.

One alternative is a hardware solution: highly reliable (or multiple) clocks with a no-break system that guarantees a continuous power supply; a universal setting mechanism through a common hardware time-broadcasting device (e.g., a satellite); etc.

Another alternative would be to avoid blocking by means of software. This alternative requires an appropriate protocol to detect maverick clocks and maintain synchronization. This subject seems of some importance if a Johnson-type model is to be built in the near future. In the next chapter (section V.1), we will discuss a scheme for clock synchronization as an extension to Johnson's model.

A very important restriction imposed by Johnson's model is the number and type of updates that are allowed. Clearly pure assignments do not suffice for all the possible applications that could make use of a distributed data base system. Investigation of the possibility of removing this strong restriction is essential for our purposes. We will thus dedicate a good part of section IV.6 to the evaluation of the possibility of extending the operations in a non-primary model (Johnson's included).

Another nuisance in this model is its profligate use of memory. Maintaining two time-stamps (T and CT) for each data base entry will certainly have repercussions on the amount of memory required.

A very nice feature of this model is the relative independence of the data base managers. If a given data base manager fails, nothing must be done by the other managers other than to stop sending messages to that particular data base manager. When the data base manager comes up again, communications restart where they were left off. In the case of a network partition (a big headache for other models), nothing special must be done. (This is true unless software clock synchronization is used and a significant clock discrepancy is present.

We will consider this problem further in the next chapter.) After the subnetworks get together communication should be established between each pair of previously non-communicating data base managers just as if a single failure had occurred.[1]

We are handicapped in evaluating Johnson's model by the fact that he does not give enough detail about the complete protocol. (For example, there is nothing on how to detect lost messages, etc.)

IV.3 Evaluation of Bunch's model

Bunch does not pretend that his model is complete. On the contrary, he points out a few of the model's problems himself; namely:

1. The speed of the algorithms is low.
2. The cost to produce and to operate the scheme is high.
3. The missing-update problem is not satisfactorily solved.
4. Unnoticed problems probably exist, since detailed analysis and implementation have not been performed.

In his first point he refers to the algorithms to handle each one of the operations and the recovery procedure. The amount of interaction between the primary and the backups is high and thus has a considerable impact on the throughput of the system as well as its cost (second point). We would add that the centralized protocol brings some additional complications. The model's throughput is fully dependent on the throughput of the primary. A high overhead in the primary computer or in the communication channels going to and from the primary would affect the system's capacity; i.e., we have a potential bottleneck.

---

[1] It should be noted that we are referring only to consistency problems. However, other types of problems could arise. For example, a user could be making decisions based upon data that is completely outdated.

The third point refers to the problem of losing updates if multiple failures occur. It is possible that all holders of a given update fail (i.e., user and primary) before it is broadcast to all other sites but after the sequence number U has been returned from the primary. If this happens, it is quite possible that a new primary will reassign the same sequence number to a different update with unfortunate consequences.

The solution that Bunch presents is that the primary should: a) send the incoming update to all the backups, together with the newly assigned number U, b) send the number U to the user, and c) signal all backups to go ahead and apply the update. The reason why Bunch does not consider this a satisfactory solution is clear; the overhead of this solution is very high and would significantly reduce the throughput of the system.

The last two items on Bunch's list of problems bring up an essential point: How resilient is the model? Unfortunately not very. As Bunch points out, if multiple failures occur during certain intervals (e.g., duirng primary recovery), the system breaks down. Network partitioning is a severe problem for this model. When a network gets partitioned into various subnetworks, we might end up with various primaries. We then have a problem of detection and correction. Detection relates to the determination that some sequence numbers have been duplicated by the various primaries. Correction involves merging the different versions of sequence numbers together. This merger is in general extremely complex. No solution is known to us. One of the problems with this merger is that updates made by one subnetwork could completely nullify others made by a second one. Example 3 of section III.1 shows the kind of logical inconsistencies that we could encounter. A possible palliative solution is discussed in section IV.4.

Some problems must be solved for this model to be operational. For example:

a. In Bunch's recovery method (algorithm 6 in [B5]) only user-controllers which are operational cooperate with a newly elected primary. Thus updates from a user-controller which is down are not taken into account even if other backup copies have already received them. Backups should probably be more active in the recovery process.

b. The backup copies journalize only after the primary tells them to do so. This happens only after the primary has found out that a backup data base manager is down. Thus, by the time the primary notifies the backups, they might have disposed of a required journal entry.

With respect to the potential set of operations that could be supported by a model of this type, there is no possible complaint. Since all the data base managers are expected to apply the updates in precisely the same order, there is no source of complication in that area. Any set of operations could be supported.

Bunch presents three types of operations: critical read, non-critical read and modifications. The critical read was introduced to support those types of applications which require an up-to-date version of the data (e.g., an airline reservation system). We should point out that the information that gets back from a critical read can not be considered up-to-date any more. As soon as the primary answers such a request it could get involved in a modification that changes the status of the data returned. In such cases some type of a test-and-set operation (or other semaphore) might be useful. (For example, we could implement an operation that adds an item to the reservation list and notifies the user whether it was successful or not.) Reliability problems (e.g., the missing update issue) make this a complex solution. What Bunch seems to have in

mind to solve this problem is to send a modification, followed by a critical read to verify the success of the modification.

It is our belief that the scheme of critical and non-critical read is unnecessarily unclear. It is not clear who is going to indicate that we want one or the other. If a user makes an update and then decides to verify its correct execution, it is clear that he wants a recent version of the data base to be the one which answers the query; but is a critical read enough? The update can be trapped in many places. The user might not get the expected result and might wonder if the update was incorrectly presented to the system or hasn't gotten to its destination. We therefore suggest that the read operations be slightly modified. Instead of two separate operations we would have one, but with a parameter (ℓu). The ℓu parameter will indicate the sequence number of the last update that must be seen before the answer is evaluated. Based on the value of ℓu and the communication status between the user-controller and the primary, the controller might decide that the primary should answer a given read (ℓu relatively recent; i.e., a critical read) or that any other copy might be queried (ℓu relatively old). This approach could improve the response time. The user-controller could maintain an ℓu parameter for each user and take appropriate action whenever a query is requested. However, this should be decided by the particular application manager.

With respect to memory requirements the overhead is relatively small. There is no need of any additional storage per record as in the other models. Only some space for communication tables (e.g., the precedence list) and the journal is required.

IV.4 Evaluation of the Reservation Center model

The Reservation Center model is in some sense a combination of the previous two models. Thus it inherited some of the advantages and disadvantages

of each of those models. The broadcasting protocol (i.e., how, where and when updates are sent) and the application strategy (i.e., resolution of conflicts by comparing tags) are similar to those of Johnson's model. The election process (for reservation center recovery) is closely related to Bunch's primary election process.

As an outcome of this inheritance, the Reservation Center model has the same limitations that Johnson's model has with respect to memory requirements and the restricted set of operations which are available. From Bunch's model it inherited the problems with partitioning (a palliative solution will be discussed later in this section) and elections. On the other hand, we avoided the inter-clock synchronization problems discussed in section IV.2 and the bottleneck potential discussed in section IV.3. The bottleneck is avoided because the reservation center's activity is much more limited than the primary's. Each period of activity of the reservation center is expected to provide enough tickets for an interval of length m, while the primary must take separate action to provide a number for each update.

The Reservation Center has a few characteristics of its own. None of the computer systems that we know of is capable of precisely reflecting the order in which things happen in real life. For example, if we have two terminals connected to a computer, it is not clear that, if two users send a message to the computer almost simultaneously, the order in which the computer will honor them will be related to the precise time they occurred. Polling order, interrupt priorities, cable length, etc. are capable of altering the real order. We thus say that there exists an uncertainty period. Given the average speeds of computers and I/O channels, we would say that the uncertainty period of the previous factors (polling, etc.) is relatively small. In Johnson's model, the clock synchronization problem causes a potentially large uncertainty

period. What has been done in the Reservation Center model is to incorporate this uncertainty period into the model. If two updates use tickets from sets generated at the same time, it is not certain (in general) which will be honored first. Thus the uncertainty period has been expanded to be at least of size m. Given that perfection is not possible (i.e., it is impossible to eliminate the uncertainty period), the disadvantage imposed by this model seems to be tolerable for small m. It is not clear that the absence of uncertainty is really a highly desired feature. A slower typist could produce similar results anyway.

In the Reservation Center model, the reservation center periodically sends a message (a set of tickets) to each data base manager and expects to receive an acknowledgment. The reservation center thus has a reasonably up-to-date knowledge of the status of the distributed data base system. This knowledge could be used to obtain a palliative solution for the partitioning problem; namely: the reservation center should not issue new sets of tickets if a majority of the data base managers have not acknowledged the reception of the last sets. (This majority could be a weighted majority in which different data base managers have different weight factors.) The inactivity of the reservation center (due to deliberate inactivity, partitioning, failure, etc.) would trigger an election process, which, if a majority of managers is available, would establish a new reservation center.

The above solution was called palliative because some applications might not be able to afford the inactivity that being a member of a minority of managers (due to partitioning or to multiple failure) would require. These managers would have to suspend their update activity until a majority is established again.

A similar technique could be used in Bunch's model. However, the primary does not have the same control over the other managers that the

reservation center does. That is, there is no interaction when the manager is inactive. An adequate extension to Bunch's protocol (e.g., some kind of "are you there?" message for periods of inactivity) would be required.

IV.5 Comparing the models in a similar environment

IV.5.1 Introduction

To be able to compare the models effectively we decided first of all to study them in an environment that could make use of any of them; i.e., in an environment that forms a least-common-denominator. We will later eliminate this restriction and deal with the inherent limitations of the models when we try to apply them in a more general environment. The least-common-denominator environment will be covered in this section while the unrestricted comparison will be left for the next one (section IV.6).

In this section we will start by defining the least common denominator (IV.5.2) and then (IV.5.3, IV.5.4), assuming that environment, try to answer several key questions. The following outline of the questions we will be examining also serves as a guide to sections IV.5.3 and IV.5.4.

IV.5.3 How good are the models during normal operation?

- IV.5.3.1 How well do they maintain the "real" update order?
- IV.5.3.2 How fast are they?
  - IV.5.3.2.1 Local application delay.
  - IV.5.3.2.2 Non-local application delay.
  - IV.5.3.2.3 System's throughput.

IV.5.4 How good are they during failures?

- IV.5.4.1 During fatal failures.
- IV.5.4.2 During non-fatal failures.

IV.5.2 The least-common-denominator environment

To set the ground rules for our comparison, we will evaluate the performance of the different models in a similar environment. This environment has been chosen to be the least common denominator of the target environments of the different models. We thus assume:

a. We are dealing with a single computer network. Thus any comparison based on the use of one particular computer system or network as opposed to any other will be ignored. For example, when we will be dealing with the different timings, we will define the time an update enters the system as the time the update was first read by a data base manager. Thus, we neglect queueing delays, operating system overhead, etc., that would be common to the three models.

b. All models are assumed to have an update broadcasting protocol that guarantees the correct and sequential transmission of updates.

c. A network delay with the characteristics described by Kleinrock [K1,K2] is assumed. Whenever specific numbers are needed to make a point, we will turn to the ARPA network data. Naylor et al. [N1] present some observed data (collected in December 73) for the ARPA network. They give the average round-trip[1] delay as a function of the number of hops[2] between the source and destination of a message. The results shown indicate that the mean round-trip delay is 50 milliseconds for 1 hop and 809 milliseconds for 13 hops (the maximum presented).

---

[1] Round-Trip means the time it takes from the moment a message is sent until the positive acknowledgment, the RFNM, is returned.

[2] A hop is the transmission of a message from one communication processor to another. The number of hops is equal to one less than the number of communications processors touched by a message.

In general the network delay time will be dependent on communication capacity, network topology, message transmission protocol, network load, etc. In any case most current networks have delays of less than one second.

d. The set of operations and the data base organization are assumed to be as described for Johnson's model. (See section III.2.)

IV.5.3 How good are the models during normal operation?

IV.5.3.1 How well do they maintain the "real" update order?

As mentioned previously, none of the models presented (nor any other that I can think of) reflects perfectly the "real" update order; i.e., the actual ordering that the updates will follow differs slightly from the real order in which they occurred. This is because of inherent characteristics of the models and because of physical properties, such as propagation delay. Rather than on perfect replication of "real" update order, emphasis is put on perfect consistency. This means that slight defects in the ordering are accepted if we guarantee the same ordering in every data base manager.

Although it does not seem to be a critical issue, we now study the uncertainty period (i.e., the probability that an update entering at time τ gets an ordering tag larger than that assigned to an update entering the system at time $\tau+\Delta\tau$).

a. Bunch's model.

In Bunch's model, the most obvious cause of asynchrony is the different network delay for different data base managers; i.e., two updates, u_1 and u_2, entering the network in that order through data base managers DBM_i and DBM_j respectively, will obtain an inverted name order if the difference in delay from DBM_j to the primary as compared with the delay from DBM_i to the primary is enough to compensate for the difference in the time u_1 and u_2 arrived.

The other end of the line, the primary, will be responsible for some added delay, but as long as it follows a first-in-first-out discipline its effect is almost null.

Finally, we have potential delay from the protocol implementation, since it is required that the order of updates generated at a single site be preserved. This problem has been eliminated by our assumption of a least common denominator.

We are left with network delay as the only source of asynchrony in this model. Thus the uncertainty interval in this model is small. That is, the probability of an update entered at time τ getting a bigger name than another update entered in a different data base manager at time $\tau+\Delta\tau$ is expected to be very close to zero for $\Delta\tau \geq 1$ second.

b. Johnson's model

It is extremely hard to establish the uncertainty period in this model. The only cause of asynchrony is the clock setting difference. It is impossible to give any universal statement without more information on the clock-setting mechanisms of the particular computer systems that we are going to be using. Using the Burroughs B-6700 computer system as an example, we realize that the clock is manually set by the operator. Furthermore, during an operating system reinitialization the clock is reset to a clock value indicated in the boot-strap deck. The operator is generally required to modify the setting. Under these circumstances, it is not hard to imagine how clocks could differ by a few minutes, and occasionally by days. (For example, the operator might forget to change the date in the boot-strap deck.)

Our only conclusion is then that the uncertainty of this model could potentially be very large. (A way to avoid very large uncertainties is

presented in the next chapter.) But it wouldn't be strange to have an uncertainty on the order of a few minutes.

c. The Reservation Center

The uncertainty period built into this model (see section IV.4) has led us to deal with this issue for the other models as well. The uncertainty is at least as big as the period m. Some minor delays (basically network transmission delays), similar to the ones discussed for Bunch's model, also exist. Given the parametric character of m, it is hard to evaluate the possible uncertainty of this model.

If we were asked to rank the models with respect to the uncertainty issue, we would probably choose the order: Bunch, Reservation Center, and Johnson. Even if for the Reservation Center the uncertainty could be as great as in Johnson's, it is a much more controlled environment. Furthermore, the values that we would typically expect for m are on the order of a small number of minutes, or even a few seconds.

IV.5.3.2 How fast are they?

IV.5.3.2.1 Local application delay

By "local application delay" we mean how fast can the update take place. This turns out to be an important issue for a distributed data base system. On many occasions a user will make reference to data he just entered or modified. In that circumstance, it is important that the data base manager he is dealing with be capable of applying updates as soon as possible.

In the Johnson and Reservation Center models, there is no reason why the corresponding data base manager shouldn't start local application as soon as possible. This is not the case in Bunch's model. In this model, the updates must be sent to the primary first and then the primary will be in charge of the

broadcasting. At least two network delays[1] are thus necessary (if the user is not dealing with the primary directly). Other delays are also present in this scheme; namely, primary queueing time, primary processing time, etc. The local application delay is therefore a potential problem for Bunch's model.

In Bunch's model the critical read could aid in this situation, but a more generalized read operation (with an ℓu parameter; see section IV.3) would be a more useful tool.

If queries and updates are treated using different queues, a similar logical race problem could occur in the Johnson or Reservation Center models. For example, a query to check if an update has been performed might be honored before the update. However, the small domain of each update (which refers only to one entry; i.e., one field) could make it acceptable to process queries and updates in a single queue and avoid this logical race.

If we were asked to rank the models again we would end up ranking Johnson and the Reservation Center first and Bunch's model last.

IV.5.3.2.2 Non-local application delay

"Non-local application delay" refers to how long it takes for all the remote data base managers to obtain the updates. Thus, non-local application delay is an important factor because it actually gives an estimate of how up-to-date the data base copies are.

Again, Johnson's model and the Reservation Center model have an easier approach. Updates are broadcast to all other data base managers as soon as

---

[1] See our discussion in IV.5.2 or Kleinrock [K1,K2] or Naylor et al. [N1].

possible. One network transmission delay is the nominal non-local application delay.[1]

In Bunch's model it takes the same delay for local and non-local application (unless we are dealing with the primary). Thus two network delays (from user to primary and from primary to backup), plus the time it takes the primary to process the update, should be accounted for.

Again our ranking would be Johnson's and the Reservation Center models first and Bunch's model last.

IV.5.3.2.3 System throughput

The amount of network traffic is similar in the three models (i.e., all data base managers get a message for each update and only one of them gets a message for any query). However, the centralized broadcasting of Bunch's model is a source of concern. There is a definite effect on the system's throughput.

To make our point clear we will work out an example. Let's assume that we have a network with three sites (s1, s2, and s3), each one of them containing a data base manager (DBM1, DBM2, and DBM3 respectively), and some other background work (B1, B2, and B3). DBM1 acts as the primary and contains the complete master copy. The other data base managers maintain their own complete backup copies of the data base. Communication is established by six half-duplex communication lines as shown in figure IV.5-a, line L_{ij} going from site i to site j (in that direction only). The capacity of these lines is assumed to be the same and will be denoted by H.

---

[1] In the three models some delay in local and destination sites should be considered, but since we are concerned only with comparing the models these delays have been neglected in all three of them.

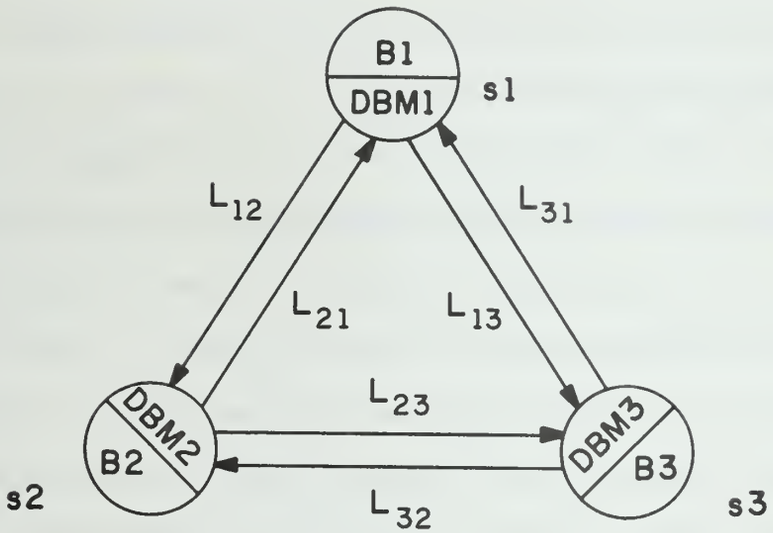


Figure IV.5-a

Graphic presentation of 3-node example. s1, s2 and s3 are computer systems, L_{ij} is the communications line between system i and j, DBMi is the data base manager located at site i and Bi represents background activity presented at site i.

Let's for a moment assume steady state conditions with a constant rate of updates entering each site and requiring .75H to be transmitted. We furthermore ignore all other existing acitivity at each site (B1, B2, B3, queries, etc.). Under Johnson's and the Reservation Center models this situation is perfectly feasible. Each line would be used at 75% its capacity. However this can not possibly be a steady state situation in Bunch's model. Using this model would require that lines L_{21} and L_{31} will be used at 75% of their capacity, L_{23} and L_{32} will remain inactive, and lines L_{12} and L_{13} will require at least 150% of their capacity! Each of the lines L_{12} and L_{13} would be required to use 75% of its capacity for updates entering at s1, another 75% for updates entered at the other manager (DBM2 for L_{13} and DBM3 for L_{12}), plus some additional traffic

in the form of names that are being returned to the corresponding manager. This totals over 150% of the line capacity, with the expected negative consequences on the non-local application delay.

Similar results are obtained with a smaller update rate if we assume that there is some activity going on between B1, B2, and B3. For example, suppose that B1 is transmitting a long file to B2 and B3. Under these circumstance DBM1 will again have trouble in using L_{12} and L_{13}. This last effect is also present in the other models but its consequences are much smaller. First, the traffic that originates at DBM1, and in general, the total line traffic is smaller. Second, lines L_{23} and L_{32} are usable and would permit DBM2 and DBM3 to obtain each other's updates regardless of activity in lines L_{12} and L_{13}. Third, DBM1 will obtain all updates from the other data bases.

To stress our point we have used high line traffic in the previous examples. We could obtain a similar situation on a larger distributed system with less traffic.

The above explanation is somewhat oversimplified but is enough to explain the relevant ideas. For example, we have neglected the network traffic that the positive acknowledgments of a reliable protocol will require.

We then conclude that, with respect to the communication lines, the throughput is potentially better in Johnson's or the Reservation Center as compared to Bunch's model.

The second factor that we will consider with respect to overall throughput is local overhead.

All the models will have some expected delay when there is local overhead due to other programs running in the same system. In all three of them we will expect a slowdown in local operations. We are concerned with the effect that high local overhead could have in the overall data base system. In

Johnson's model, high overhead in a data base manager causes practically no symptoms at a second data base manager. Communications from the overloaded data base manager will be slowed down (as well as local update arrival, if a positive acknowledgment is required), but this is about all the effect. This statement is valid for all data base managers other than the primary in Bunch's model, and all but the reservation center in the Reservation Center model.

The Reservation Center sensitivity highly depends on the value of the parameter m that is chosen. If m is large the reservation center will seldom go to work and probably the effect will pass unnoticed. However, if m is small the slowness of the reservation center will certainly be noticed since the data base managers will start exhausting their ticket supply and sending messages to the reservation center asking for more. This will make the reservation center increase the allotment of tickets per emission which will probably stablize the situation (but using an "effective" m different from that planned.)

Bunch's model is much more sensitive to high local overhead at the primary site. Actually the overall update throughput of the complete distributed data base system is highly dependent on the primary performance.

We then conclude that the ranking with respect to potential throughput is: Johnson, Reservation Center (very close to Johnson's) and Bunch.

IV.5.4 How good are they during failures?

The study of failures is a broad and fertile field for research. There are many combinations to take care of. It is difficult to be certain that we have enumerated all of them. We will therefore consider broad classes of failures, and not concern ourselves with all possible events that could cause trouble.

We then classify failures as follows:

1. Fatal failures.
   a. Single failures.
      i. In hosts and minihosts
      ii. In the network
   b. Multiple failures, partitioning.
2. Non-fatal failures.

IV.5.4.1 Fatal failures

This type of failure is characterized by the logical inactivity of the element that fails.

a. Single failures
   i. In hosts and minihosts.

A fatal failure causes the corresponding system to suddenly stop any activity. There are two types of problems, internal and external. Internal problems refer to the damage that a system causes to itself and its users. This is an important problem in many current computer installations. Techniques to avoid substantial damage from such occurrences are widely known (Martin [M1]) and include redundant hardware, journalizing, backups, check points, etc. Our distributed data base study does not give any more insight into these problems. We will thus neglect the problem, and expect the appropriate use of resiliency techniques in the data base managers' design.

External problems are a typical example of a new complication introduced by a distributed data environment. One must establish what a given data base manager (DBM_i) should do when another manager (DBM_j) fails. The obvious step for DBM_i would be to stop any further communication with DBM_j as soon as it

is aware of the failure. The three models behave similarly under these circumstances and we won't enter into the details.

In Johnson's model this is practically the only external problem that must be taken care of. In the remaining two models there exists an element of discord. All but one of the possible system falures would cause only the problem mentioned. But in the case of failure of the primary in Bunch's model or of the reservation center facility in the Reservation Center model, we have some additional aspects to be covered.

Bunch [B5] describes an election protocol to be carried out when a failure of the primary is detected. (See section III.3.4.) The process is started as soon as a failure is detected (probably through a time-out of some network message establishing an unsuccessful message delivery due to a dead host). The election process involves an exchange of messages between all primary candidates and all other data base managers.

A few of the disadvantages of having to have such a procedure are:

a. The election process is started at the worst moment, when some data base manager is waiting for the primary to return a tag. That data base manager will necessarily have an additional delay for one of its updates.

b. The primary recovery procedure presented by Bunch ([B5] Algorithms 9 and 10) are costly in time. After the new primary is elected, it has to establish its working situation: to obtain all pending updates from all the user sites and to establish if names have or have not been assigned, to process any pending updates, etc.

c. There is no provision for any unique order for the updates which have not gotten a name. This would greatly increase the uncertainty period

described in section IV.5.3.1. Notice that points A and B also affect the uncertainty.

d. The protocol should be enriched with a safeguard against failure while the primary is recovering. Given that there are a few moments where there is no flow expected toward the primary (i.e., the primary has all the facts and is processing them), some time-out would have to be established for detection of such a failure.

The Reservation Center model has a corresponding problem. When the reservation center fails, a similar election must be carried out. There are, however, some clear differences:

a. Since at least one extra set of tickets is available at each data base manager, failure of the reservation center will probably be detected before all the local tickets are exhausted. Thus no updates will incur an additional delay to affect the uncertainty period.

b. The new reservation center does not need to know precisely the terminating situation of the last reservation center. Obtaining an estimate of the highest ticket it generated would be enough. Thus the time required for the new reservation center to start operation would be rather short. (This estimate could be used in the election process, so it would be known to the elected reservation center.)

c. Since the updates are expected to keep flowing while the recovery is going on, no measurable difference would be expected in the uncertainty period as discussed previously.

d. Detection of failure during recovery should be added to the protocol, as in Bunch's case.

ii. Single fatal failures in the communications network.

Single fatal failures in the network could occur in the communication lines or in the communication processors (CP). The effect of a fatal failure of a communication line varies with network topology. In the ARPA network all CP's are at least two-connected, and a failure in one communication line will at most affect the network delay.

If the topology is star-like or ring-like, the consequences could be more severe. In these cases, the magnitude of the failure must be considered (i.e., whether the failure was only in one direction or in both, etc.). These cases will not be dealt with directly in this thesis. Other cases discussed will exhibit behavior similar to these.

Failure of a CP produces for all practical purposes the illusion of a system failure, with basically the same external problems discussed earlier. However, a completely new internal problem occurs. The data base managers attached to the CP find themselves isolated from all of the network. They suffer the illusion that a multiple failure affecting all other systems occurred. This is the simplest case of partitioning, which will be discussed below.

If we have to rank the models with respect to their sensitivity to single fatal failures, we would certainly put Johnson's model as the most insensitive one. The Reservation Center would go next because of the simpler election protocol. Finally, Bunch's model is considered to be the most sensitive of the three.

b. Multiple failures

Multiple failures refer to multiple occurrences of simple fatal failures. In many cases, the consequences of the effect of a multiple failure can be obtained by simple addition of the effects of the individual failures.

But in other cases, there will be an additional problem due to the multiplicity. We next deal with one such case, partitioning of the network.

Bunch [B5] discusses the problem but does not give any solution. In section IV.4 we presented a possible solution, a minority partition (an isolated set of less than half of the data base managers, according to some weighted criterion) stops any update activity. In the discussion of Johnson's model (section IV.2) we explained how this model has an advantage with respect to this problem.

If we have to rank the models with respect to their insensitivity to multiple failures we would have to start with Johnson's model, followed by the Reservation Center and Bunch's model in that order. This ordering follows the increasing complexity of their behavior during network partitioning.

IV.5.4.2 Non-fatal failures

Non-fatal failures are difficult to handle. There are generally many manifestations of such failures with different grades of difficulty for detection and prevention. If a component or system detects that another component or system is failing, it will be essential to establish which one is really failing - the detected or the detector. Some redundancy or double checking would be helpful for such cases.

Once a component or system is detected to be failing, we should consider how to keep it from causing more trouble. As Bunch [B5] says, how do we make him commit suicide? The failing part may not be willing or capable of committing suicide. Due to the difficulty involved, none of the models directly attacks this area. We could optimistically expect such failures to produce inconsistent information which is easy to detect and discard (i.e., by parity detection, etc.). As mentioned before, we rely on the conventional techniques

(parity checking, resiliency protocols [D1], etc.) to assist us. However we are never certain that a failure did not escape detection.

IV.6 Overall power of the models

In the previous section we tried to compare the models under a very strong assumption; namely that the environment was a least common denominator that could make use of any of the given models equally well. The purpose of such a restrictive assumption is clear when we realize the different characteristics of each model. What we have done, however, has been to restrict the power of the models in certain respects to suit the common environment. In this section we will eliminate this restriction and tackle the job of exposing the real power of each one of the models. By "power" we will mean the collection of operations that a given model is capable of supporting. Immediately we know that Bunch's model is very powerful. Given that all the data base managers are expected to apply all the updates in precisely the same order, there is no restriction on the type of operations that the model is capable of supporting. The same comments are not valid for the original presentation of the other two models (which we will now call non-primary models). We question such a design restriction. In this section we will determine whether and to what extent we can relax this restriction.

In section IV.6.1 we will build up a series of definitions and concepts required for our future discussion. We will end up with some idea of how to extend the set of operations of a non-primary model. In section IV.6.2 we will discuss some complications which would arise if we actually tried to implement a non-primary model with an extended operation set.

IV.6.1 Concepts and definitions

IV.6.1.1 Notation and basic definitions

We start with some notations:

An update u is denoted as a triple (op, p, tag), where op is an operation type, p a set of parameters for the given operation, and tag is the synchronization tag used in any of the given schemes. The p factor is further defined as a pair (ℓ, v) where ℓ is the parameter (or parameters) used to identify the location where the operation is going to take place, and v is the set of values required by the given operation (e.g. the new value to be assigned, or an increment to be added). P will denote the universe of the set of parameters p.

A data base db is defined as a collection of bits. Given that data bases are expected to change dynamically with the application of updates, we will denote the state of the data base after i operations have been performed as db_i, with db_o being the initial data base. The universe of data bases will be denoted as DB.

We now start with our definitions:

1) A data base system S is defined as a tuple (DB, P, db_o, op_1, op_2, op_3 ... op_n) where db_o is the initial data base for all data base managers, DB the universe of possible data bases, P the universe of possible parameters, and op_1, ..., op_n are the operations available in S.

2) An operation is defined as a function op: $PxDB \rightarrow DB$

i.e. $op(p, db_i) = db_{i+1}$

3) An operation is said to be one-reversible in S if and only if for any p and db_i there exists an operation op^{-1} in S such that

$$db_i = op^{-1}\ (p,\ op\ (p,\ db_i)) = op^{-1}\ (p,\ db_{i+1}) = db_{i+2}$$

4) An operation op_1 is said to be n-reversible in S if and only if there exists an operation op_1^{-1} in S such that for all $op_2,\ldots,op_n$ and all possible corresponding $p_1, p_2,\ldots,p_n$,

$op_1^{-1}(p_1, op_n(p_n, op_{n-1}(p_{n-1},\ldots, op_2(p_2, op_1(p_1, db_i))\ldots)))$

$= op_n(p_n, op_{n-1}(p_{n-1}, op_{n-2}(p_{n-2},\ldots, op_3(p_3, op_2(p_2, db_i))\ldots))).$

5) A journal entry for operation $op_i(\ell,v)$ is defined to be a pair (ℓ,v') where ℓ is the same location indicated by the operation and v' is the value of the location prior to the application of the operation.

A system is said to have a modification journal if the journal entry for each operation that is performed is appended to a file called the journal.

A system is defined to be journal reversible if it has a modification journal. An operation op_i is reversed by taking v' from the corresponding journal entry and storing it at location ℓ.

6) An operation is said to be naturally reversible if it's n-reversible for all values of n.

If an operation is naturally reversible or journal-reversible, then it is simply called reversible.

7) Two operations op_1 and op_2 are said to be commutative if and only if for all db_i and all p_1, p_2,

$op_1(p_1, op_2(p_2, db_i)) = op_2(p_2, op_1(p_1, db_i))$

8) If op is commutative with itself (i.e. $op_1 = op = op_2$ in the previous case), then we say that op is commutative.

9) An operation op is said to be homing if and only if

$op((\ell,v), db_{i+1}) = op((\ell,v), op'((\ell,v'), db_i))$

for all possible operations op' in S, and all db_i. (ℓ is a fixed location and v and v' are arbitrary parameter set values.)

Example 1: Assign (ℓ, v)

The assign operation stores the value v at location ℓ regardless of its previous value.

Assign is not one-reversible: given that we just performed an assign there is no way of knowing the previous value (destroyed by the assign) by just knowing the update parameters. Assign is not commutative: if we have two sets of parameters (ℓ, v_1) and (ℓ, v_2) the value of location ℓ differs if the order is exchanged. That is, it is v_1 after assign ((ℓ,v_1), assign ((ℓ, v_2), db_i)) but is v_2 after assign ((ℓ,v_2), assign ((ℓ,v_1), db_i)). Assign is a homing operation; regardless of the previous values a location ℓ will have the value v immediately after the operation assign ((ℓ,v), db_i) is performed.

Example 2: Iadd (ℓ,v), Isub (ℓ,v).

The Iadd (Isub) operation adds (subtracts) the value v to (from) location ℓ. We assume that v and the contents of location ℓ are both integers.

Integer addition is reversible in S(DB,P,db_o, Iadd, Isub). It's sufficient to subtract v rather than adding to obtain the desired result; i.e. Isub = $Iadd^{-1}$. (An alternative system could use only Iadd by defining $Iadd^{-1}(\ell,v)$ = Iadd(ℓ,-v).) Integer addition is also commutative in S, as is known from standard algebra. Iadd is not n-reversible in S(DB,P,db_o,assign, Iadd, Isub) for n>1. Once an assignment occurs it is impossible to undo any Iadd, and we haven't assumed any way of knowing if such an assignment has occurred or not. However, Iadd is one-reversible in the same system.

IV.6.1.2 Operation power

The operation power is defined according to the number of operations required to modify any one field of a given logical record. An operation will be called a micro-operation if it must be used in conjunction with other micro-operations to modify a single field, a mini-operation if it alone can modify a

field, and a maxi-operation if it can modify more than one field (in one or more records).

A classical example of micro-operations arises in updating a chained element. The series of updates indicating every single change required to unchain and re-chain the modified element would be called micro-operations. Generally speaking, the use of micro-operations is not advisable in a distributed data base environment. This is because after the application of a micro-operation the data base could be left in an inconsistent situation (in the middle of changing a pointer, for example). Some locking mechanism must then be used to prevent interference between updates. If the application requires such micro-operations, it seems reasonable to extend the update to a collection of micro-operations packed together and specified to be uninterruptible. In this case we could as well reclassify this update as a mini- or maxi-operation, as appropriate.

In the next category, mini-operations, we assume that only one update is required to modify any single value. This does not imply that chained elements are not permitted. The local data base managers are expected to have enough intelligence to infer such modifications, or the update is expected to contain a series of micro-operations which will have to be done in an uninterrupted sequence. (Implicit or explicit locking might be required.) As an example of updates in this category, we mention the ones used in Johnson's model [J2]. The assign (ℓ,v) operation described previously is a mini-operation when ℓ identifies exactly one logical record.

Finally, maxi-operations have the power to affect more than one record at a time. Many examples can be found in the relational data base literature. One specific example is an update that changes all green elements to blue. This update could also be performed as a collection of mini-operations, each one

indicating the change in the color field for a single record. However, this is an extremely dangerous solution with side effects similar to those mentioned for micro-operations. Suppose that while we are changing the individual elements from green to blue we get another update indicating a change of all blue elements to red. We could then turn out with some of the green elements not converted to red, since when the second update arrived they were still green. Again, some implicit or explicit locking mechanism is required to ensure consistency.

IV.6.1.3 Theoretical systems

S1. Single homing, non-reversible operation. We start with a system S1(DB, P, db_o, op1) where op1 is assumed to be a homing, non-reversible operation. (The assign operation mentioned earlier is a good example for op1.) The reader should notice that we use op1 to indicate an operation type, while op_i earlier denoted one of a sequence of operation applications.)

Given that op1 is not commutative, a precise and unique order must be established. Synchronization tags or any other mechanism that guarantees this order is vital for S1 to be feasible. If all the op1 operations can be ordered for application at the different sites in precisely the same order (Bunch's model), there is not much left to say other than that each data base manager is expected to perform them in that precise order. If, on the other hand, arrival order is arbitrary but tags are available to sequence the operations uniquely (Johnson's or the Reservation Center models), we can handle S1 if op1 does not modify the field or fields that are used to select the records.

In this case we can store the tag (G) associated with each instance of op1 together with the field it modifies. At the arrival of an update u, the tags can be compared. If the tag of u is smaller than G, then we can forget about the update u. The homing property of op1 guarantees that we will end up

with the same stored value as if u had been applied in its proper sequence. If, on the other hand, the tag of u is bigger than G, we perform the update and replace G by the tag of u.

If op1 modifies the field(s) that are used to select the records some problems arise. For example, let's assume that op1 is a maxi-operation. The maxi-operation op1(ℓ,v) is expected to have the power to refer to more than one record. This could be done explicitly (ℓ = set of addresses) or implicitly (ℓ = description of characteristics). If we use implicit locations we could have problems during the application of updates. For example, if an update u_1 calls for all things that are green to be changed to blue, and update u_2 requests that a specific part (PA) be changed to green, then the outcome depends on the order in which the updates are applied. If u_1 is executed first, PA will turn out to be green; otherwise it will come out blue. A similar situation will occur when explicit locations are used (i.e., the maxi-operation resembles a collection of mini-operation pasted together), but in this case problems arise in the emission of updates. The emitting data base manager will have to wait until all pending updates arrive, or risk that some of the missing updates will effect changes that would have modified the update. It should be noticed that not waiting will not cause a consistency problem in the data base (i.e. all copies will eventually be consistent, as opposed to what happens when implicit locations are used), but the data might turn out to misrepresent reality. Waiting is not realistic since we could end up with all the data base managers waiting for the others to act. The problems that arise when operations are allowed to modify fields which are used by other updates to select records will be more extensively discussed in section IV.6.2.

S2. S1 plus delete and create. One of the restrictions of S1's definition is that its size has been established to be static; i.e., there is no

provision to create or delete records. We now solve this defect by defining S2. To obtain the system S2(DB,P,db_o,opl,create,delete) we have added to S1 two additional operations, a create operation which generates new records for the data base, and a delete operation which eliminates them. Basically all comments about opl in S1 are valid for S2 as well.

The create operation could be made explicit or implicit as discussed in section III.2.2. In this latter case, S2 is reduced to S2(DB,P,db_o,opl, delete).

Since creation does not commute with opl, the order of application is relevant. Furthermore, creation is not itself commutative, and we haven't established what should be done when two creations for the same location arrive. It is not clear that ignoring a second create operation is the right decision. Doing so could simply mask an error. Alternatively, we could avoid the occurrence of two creates by two means. i) Eliminate the possibility of two sites generating create operations by restricting this function to a single data base manager. (This introduces some problems when the creator data base manager is not available.) ii) Eliminate the possibility of two sites generating a create to the same location by preassigning the locations that each data base manager could generate. The remaining problem is with respect to the non-commutativity of opl and create. What should be done when we get an opl for a nonexistent location? The various alternatives include: i) wait until the create arrives, or ii) perform an implicit create, but flag the given record until the create arrives. When the create arrives, just eliminate the flag. The homing characteristic of opl makes the initial value of that location irrelevant. The user would be expected to be notified of the flag.

The delete(ℓ) operation is assumed to eliminate the location(s) indicated by the ℓ parameter from the data base. Delete is clearly not

commutative with any other operation. This introduces potential hazards. If delete is incorrectly executed before the corresponding create, the record won't be deleted as expected. Similarly, if delete is incorrectly executed before an opl operation, we would either end up with the record staying alive (due to implicit create), or an eternal wait (if we follow the advice of waiting for the create to arrive). These comments as well as the proper solutions were discussed in section III.2.2 when we spoke of the delete operation for Johnson's model.

At this moment Johnson's model can be explained in a few words. His model is an S2(DB,P,db_o,assign,delete) system with operations that do not modify the selecting field. It has implicit creates and deletes handled as described in the previous paragraphs.

S3. A single commutative, non-homing operation. We now study the system S3(DB,P,db_o,op2), where op2 is a commutative, non-homing operation. This system is probably the simplest one possible. In S3 the order becomes irrelevant and no synchronization tags nor any master-slave relation (Bunch) is required or useful. Mini-operations or maxi-operations could be used. The only possible problem would be if we lose an update. Resiliency protocols [D1] could be used to take care of this hazard.

Data base managers would be expected to perform updates as they get them. Usually the application delay will be small. But if rapid application is not important, the data base managers could be allowed some flexibility. We still have a slight complication. Again, as in S1, we are trapped with a fixed file size. There is no provision for creations and/or deletions. This leads to S4.

S4. S3 plus create and delete. S4 is defined as S4(DB,P,db_o,op2, create,delete), with op2 as in S3 and create and delete as in S2.

Once more the problem is that the create and delete operations do not commute with the standard operation (op2). The comments about create that were made for S2 are valid here if the create always initializes values to a standard value (zero for standard algebra). For example, Iadd(ℓ,10) will create a record at ℓ with value 10 only if create(ℓ) creates a record at ℓ with value 0. Otherwise some waiting for the create might be required. On the other hand, delete is hard to implement without tags. If we implement S3 and S4 without synchronization tags, we cannot pull the same trick we used for S2. When we get a delete, we don't have any way of knowing if updates arriving at about the same time should go before or after the delete. Some alternatives could be: i) never to recreate deleted locations (i.e., we mark the record as deleted and leave it there forever (or for a long while); or ii) to use synchronization tags and solve the problem with the a posteriori idea of S2 and Johnson's model, i.e., the utilization of Johnson's A matrix to establish when a delete has been seen by all the data base managers.

S5. Combination of S1 and S3. We will now deal with S5(DB,P,db_o,op1, op2), where op1 is defined as in S1 (and S2) and op2 as in S3 (and S4). In this system there are two classes of updates, one for op1's and another for op2's. If, for any given record, we get only elements of one class, we could treat the updates as in S1 or S3 (as required). New complications arise when we mix the classes. Given that op1 and op2 do not commute, the order of application is again relevant. If we incorrectly perform an op1 before an op2, it will seem that we are in trouble since op1 is not reversible. This is fortunately not the case. We just have to remember that op1 is a homing operation. An operation which should have been performed earlier can just be ignored. That is, we ignore the op2 and reapply the op1. On the other hand, when an op2 operation is incorrectly performed before an op1 operation, we have some minor problems. We

have to exchange the order. This would seem to imply that we have to reverse op2, but this is not true. Given that op1 is homing, we could forget about undoing the op2 and just reapply it after the op1.

In summary, for each given field in the data base, we could have two tag fields (G1 and G2), one for the tag of the most recent operation of each class. When an op1 operation arrives we will compare its tag with the G1 tag and act just as we did in the S1 system. If the incoming update has a tag smaller than G1, we forget about it; otherwise we apply the update and store its tag in G1. In this latter case we still have to consider the possibility of an op2 operation that arrived previously having a tag bigger than the new G1. To do this G1 and G2 are compared. If G1 is bigger we are done; otherwise we should look for all possible op2's that should be reapplied. (This implies the existence of a journal where updates are saved for future use. In this case if that G2 is bigger we know that there exists at least one entry in the journal which should be reapplied). In order to make this last operation not too slow a chained list might be used. When an op2 arrives things are simpler. If its tag is bigger than G1, it should be applied; otherwise ignore it. Under this scheme op2 operations might have to be applied more than once. If this is a big problem, something like a delay pipeline might be advisable to minimize the probability of having to redo op2's. (See section V.2.)

If operations which modify the selection field are used, then the implicit or explicit location problems discussed in S1 are still with us.

Once more we have ended up with a static file size, but this time we will solve it by just saying that the solution is similar to the one given in S2 and S4.

S6. A three-operation system. Finally, we will study the system S6 $(DB,P,db_o,op1,op2,op3)$ where op1 and op2 are as before, and op3 is a commutative

non-homing operation which does not commute with op2. With trivial changes this system could be extended to S(DB,P,db_o,op1,op2,...,opN), where op1 is as before and op2,...,opN are commutative, non-homing operations which do not commute with each other.

In S6 we have three classes of operations as compared with two in S5. In general the interaction between op1 and the other two will be similar to that in S5 (i.e., we now have 3 tag fields, G1, G2 and G3, and all checking mentioned for G2 will have to be repeated for G3). However the interaction between op2 and op3 is important. If the application order at the different sites is not identical, then we require that op2 and op3 are either one-reversible or journal reversible. We have to actually deal with problems of undoing updates that were done in a wrong order. S6 must then be extended to include either a journal or the inverse operations $op2^{-1}$ and $op3^{-1}$. (This does not necesarily imply new operations. For example, for Iadd we could use $Iadd^{-1}(\ell,v)$ = Iadd(ℓ,-v)). When an op2 or op3 operation is found to have been performed out of sequence it should be undone, and the correct sequence should be applied. Otherwise the data bases will turn out to be inconsistent. (Repeating example 2 of section III.1, if we have op2 = increment by given amount and op3 = increase by the given percentage of current value, the order is relevant. The operations op2(ℓ,10) and op3(ℓ,10) will give different results if not applied in the same order).

There is still an outstanding question, how far to undo? If we are dealing with mini-operations the problem is greatly reduced, since we just have to undo operations that affected the same field as the out-of-order update. With respect to maxi-operations, it is not clear whether that's enough. We again have problems, according to the way the locations to be affected are indicated in the update (implicitly or explicitly) and we refer to section IV.6.2 for the corresponding discussion.

Here more than in any other system there is a potential for high overhead for all the undoing and redoing that could take place. A feature (delay pipeline) that could be used to minimize the probability of undoing is discussed in section V.2.

In most of the remaining aspects (deletions, creations, scheme to apply opl operations, etc.), S6 is similar to S5.

IV.6.2 Evaluation of the feasibility of the addition of operations to a non-primary model

IV.6.2.1 Extension of operations in a non-primary model

In a non-primary model we find ourselves in an environment in which different data base managers experience different update application orders (as opposed to Bunch's model.) The tags were designed to enable the data base managers to establish when a violation of the update order has occurred and to take the appropriate steps to resolve this conflict. In Johnson's model this conflict is easily resolved due to the restricted set of operations available. However, an easy resolution is not encountered in the more general systems discussed in the previous section.

An initial solution to the conflict could be, upon arrival of an out-of-order update, to undo all updates that were applied in the wrong order. This undoing represents a possibly substantial increase in processing times and costs for updates. In order to minimize this undesired overhead, a pipeline delay will be studied in section V.2. An arriving update would not be immediately applied. Instead it would enter one side of a pipeline and exit through the other side after a delay d. Only updates coming out of the pipeline would be applied, but before application of any update the pipeline would be examined to locate all other updates that should precede it and they would be applied first. The use of a pipeline delay certainly reduces the probability of undoing, at the

expense of higher local application delay. (See section IV.5.) It should be pointed out that by using a reasonable pipeline delay, we avoid the bulk of the undoing process, but certainly not all. If a network partition occurs or a host or hosts go down holding the only copies of a given update, it is very likely that very old updates will arrive at each data base manager. In those circumstances a very high undoing cost would have to be paid, given the large number of updates that would probably require this undoing process.

In view of the above comments, it seems reasonable to study whether it is possible to transform (reevaluate) an incoming out-of-order update u_o into a set of alternative updates u'_{oj} in such a way that we will obtain the same results that we would have obtained if the updates had arrived in the right order.

We will start with the simplest case: we will assume that we are dealing with two updates u_o and u_1 that should follow that order. However, we assume that for the data base manager that we are dealing with the arrival and application orders have been inverted; i.e., u_1 has arrived and has been applied to the data base before u_o's arrival. The data base is assumed to consist of a set of records, each record having a number of fields. Each update u_k is of the form "For all the records in the data base which satisfy condition C_k modify fields F_k as follows E_k". The fields of our data that are tested by C_k (i.e., are variables in the Boolean expression C_k) will be denoted by C_k^F. We will refer to the instant at which u_1 arrives as τ-1, and the instant at which u_o arrives at τ. (u_1 has already been applied at τ.) Finally, the instant at which u_o has successfully been applied is denoted τ+1. (See figure IV.6-a).

It is clear that condition C_k holds for different records at different instants. We will then call the set of records for which C_k is true the domain of C_k, denoted at an instant I as $D_I(C_k)$.

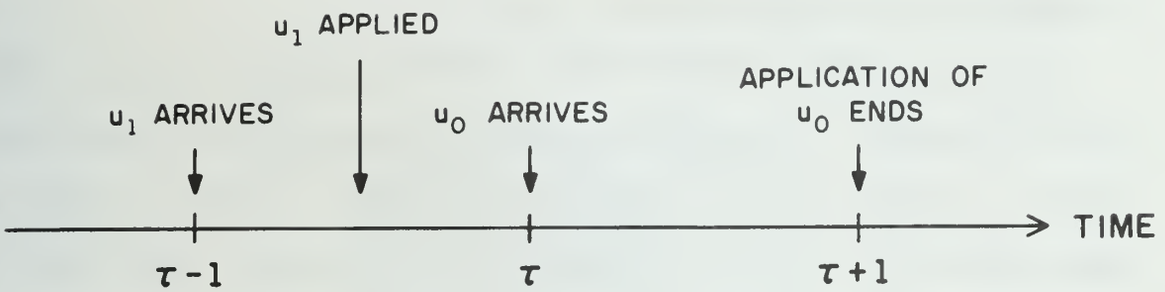


Figure IV.6-a

A time line describing the order of the events discussed in the text.

An example at this point might be helpful. Suppose that the data base consists of a set of personnel records, each record having fields for name, SSN, age, salary, and job class. Suppose update u_1 reads

"For all records such that job class = typist, increase salary by $1000."

In this case, F_1 consists of the single field salary. C_1^F consists of the field job class. $D(C_1)$ is the set of all records for which job class = typist.

Finally each update u_k will be associated with a tag t_k ($t_o<t_1$). As discussed previously (sections III.2 and III.4), we expect that each field of every record will have a tag space. Whenever an update u_k affects a field of a given record, the tag attached to that field is changed to t_k.

For our discussion we will (case 1) first assume that $F_1 \cap C_o^F = \emptyset = F_1 \cap C_1^F$. That is, the application of u_1 does not affect the fields which C_o and C_1 use to establish their domains. This implies that $D_\tau(C_o)=D_{\tau-1}(C_o)$ and

$D_\tau(C_1)=D_{\tau-1}(C_1)$. Looking again at our example, we see that $C_1^F \cap F_1=\emptyset$ is satisfied in that case. Now suppose update u_o reads

"For all records such that salary > \$10,000, decrease salary by \$2000."

Here $F_o=C_o^F=$ salary , and $F_1 \cap C_o^F=$ salary $\neq\emptyset$. We immediately see the problem. If update u_1 is applied (out of order) to Joe's record, and Joe is a highly paid typist, making \$9500, then the subsequent application of u_o would cause Joe an unfortunate net loss of \$1000. What has happened is that Joe's record shouldn't have been in the domain of u_o, but if u_1 has been applied by the time τ when u_o arrives, then $D_\tau(C_o)$ includes poor Joe (or, more precisely, his record).

The assumption $F_1 \cap C_o^F=\emptyset=F_1 \cap C_1^F$ simplifies our initial study but will be removed later (case 1B). Adding the restriction $F_o \cap C_o^F=\emptyset=F_o \cap C_1^F$ leads us to a Johnson's type model, which will be studied as case 1A. Finally we will expand our study to include three or more updates (case 2).

Case 1 - Assuming $F_1 \cap C_o^F=\emptyset=F_1 \cap C_1^F$

The data base can be subdivided into six regions (figure IV.6-b) with the following characteristics:

Region 1. This region contains the records which belong to $D_{\tau-1}(C_1)(=D_\tau(C_1))$ and do not belong to $D_\tau(C_o)$ or $D_{\tau+1}(C_o)$; i.e., those records which are affected only by u_1.

Region 2. This region contains the records which should be affected by u_1 but do not belong to $D_\tau(C_1)$; i.e., the applicability of u_1 is only detected after applying u_o. This implies that for these records u_o modifies some fields which affect the fields tested by $C_1(F_o \cap C_1^F \neq\emptyset)$.

Region 3. This region is basically the inverse of region 2. It contains all the elements which belong to $D_{\tau-1}(C_1)$ but which would be eliminated from the domain of u_1 if u_o had been applied first. As before this implies that $F_o \cap C_1^F \neq\emptyset$.

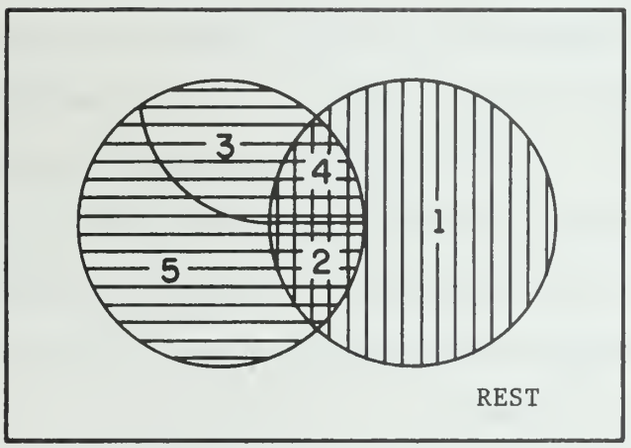


Figure IV.6-b

Partition of the data base into six regions (1 to 5 and Rest). Records in left circle (horizontal stripes) should have been affected by u_o and those in right circle (vertical stripes) by u_1, if the updates have been applied in the proper order.

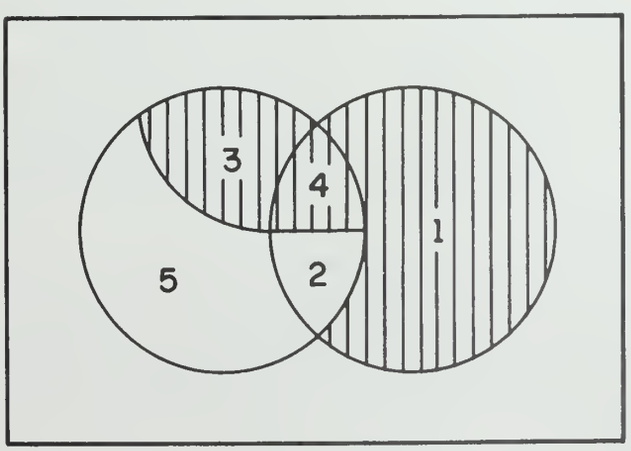


Figure IV.6-c

Situation at instant τ. Vertically striped area indicates $D_{\tau-1}(C_1)$.

Region 4. In this region we place all records that should be affected by both updates. Thus region 4 contains all records which are in $D_\tau(C_1)$ and in $D_{\tau+1}(C_1)$, as well as in $D_{\tau-1}(C_o)$. That is, the fact that u_1 has been applied before u_o has no effect on the applicability of both updates to these records. But the result of applying the updates may be affected by the inverted order. This is the "classical" conflict area.

Region 5. This region is symmetrical to Region 1, but for u_o. It contains all the records which should be affected only by u_o and for which the application of u_1 is irrelevant.

Region Rest. Records affected neither by u_o nor by u_1, no matter how they are applied. It is irrelevant in our discussion.

We next present algorithm A, which will generate the updates u'_{oj}. We do not make any statement about the optimality of this algorithm. This algorithm is to be applied at time τ, i.e. upon arrival of u_o.

Algorithm A

a. If $F_o \cap C_1^F = \emptyset$ then (Region 3 is empty) generate:

u'_{01} = "For all records in the data base which satisfy $C_o \cap C_1$, solve conflict" (covers region 4)

else generate:

u'_{02} = "For all records in the data base which satisfy $C_o \cap C_1$ undo u_1" (covers region 3 and 4)

b. Generate:

u'_{03} = "For all elements in the data base which satisfy $C_o \cap$(C_1 not applied) modify F_o". (We apply u_o to regions 2 and 5 and if u'_{02} was done then to regions 3 and 4 as well.)

c. If $F_o \cap C_1^F \neq \emptyset$ then generate:

u'_{04} = "For all records in the data base which satisfy $C_o \cap$(C_1 not applied) modify F_1" (we apply u_1 to regions 2 and 4.)

Explanation:

We have used the condition (C_1 not applied) a couple of times. This means that for the current state of the record, C_1 has not yet been applied. This could be tested by simply choosing any field F* of C_1^F and checking whether its related tag is t_1; i.e., (C_1 not applied) is equivalent to (F*'s tag $\neq t_1$). The "conflict solving" of u'_{01} is assumed to set the tags of C_1^F for region 4 to read "t_1". Similarly, the "undoing" of u'_{02} must also reset tags from t_1 to their previous value.

If $F_o \cap C_1^F = \emptyset$ then the application of u_o can't possibly add or subtract elements from the domain of C_1, and thus all the elements that must be affected by u_o only, u_1 only, or both of them, are perfectly known. In this circumstance, u'_{01} will handle the "both" case and u'_{03} the "u_o only" case ("u_1 only" has already been taken care of at τ). In u'_{01} we vaguely say "solve conflict" because the steps to be taken are a function of the type of operations that we have. In general the conflict could be solved by "undo u_1, apply u_o, reapply u_1". But this might be too extreme for some cases. For example, if u_1 is implementing a homing operation (e.g. assigning) then the conflict could be solved by ignoring the modifications of fields already touched by u_1. If we have that u_o and u_1 are commutative (e.g. integer addition) then u_o can simply be applied after u_1.

The above comments are not true for the case when $F_o \cap C_1^F \neq \emptyset$. Under this condition it might well be possible that u_o modifies something that would

preclude the applicability of u_1 (i.e. region 3 is not empty). Thus all the records in $C_o \cap C_1$ will have to be tested to establish if they belong to region 3 or 4, and in general (e.g. for assignments) u_1 will have to be undone tc be able to establish this situation. This has motivated the u'_{02} update.

u'_{03} is intended primarily to cover regions 2 and 5. However, if u'_{02} is applied, it will also have an effect on regions 3 and 4.

If $F_o \cap C_1^F = \emptyset$ we are done after step b, since Region 2=Region 3=$\emptyset$. Otherwise we still have to consider the application of u_1 to region 2 and, because of u'_{02}, to region 4. u'_{04} will serve this purpose.

Case 1A. Assuming $F_1 \cap C_o^F = F_1 \cap C_1^F = F_o \cap C_o^F = F_o \cap C_1^F = \emptyset$

This turns out to be a logical simplification of case 1. (Note all the conditions involving $F_o\ C_1^F$ in the previous case.) Under these conditions we have that regions 2 and 3 disappear and figures IV.6-b and IV.6-c get transformed to figures IV.6-d and IV.6-e respectively. Algorithm A simplifies to algorithm A1:

Algorithm A1:

a. generate:

u'_{01} = "For all elements in the data base which satisfy $C_o \cap C_1$ solve conflict" (covers region 4)

b. generate:

u'_{03} = "For all elements in the data base which satisfy $C_o \cap \overline{C}_1$ modify F_o" (We apply u_o to region 5.)

If we further assume that all we have are homing operations (e.g. assignments) we simplify things even more since "solve conflict" becomes "do nothing" in u'_{01}, i.e. we discard u'_{01}.

It is interesting to note that Johnson's model satisfies precisely these simplified conditions.

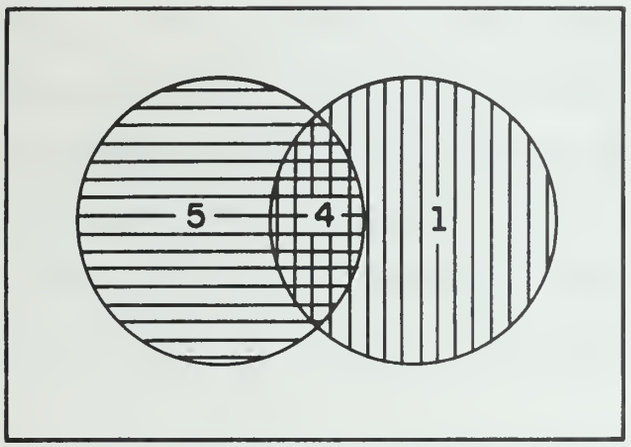


Figure IV.6-d

Equivalent of figure IV.6-b for case 1A.

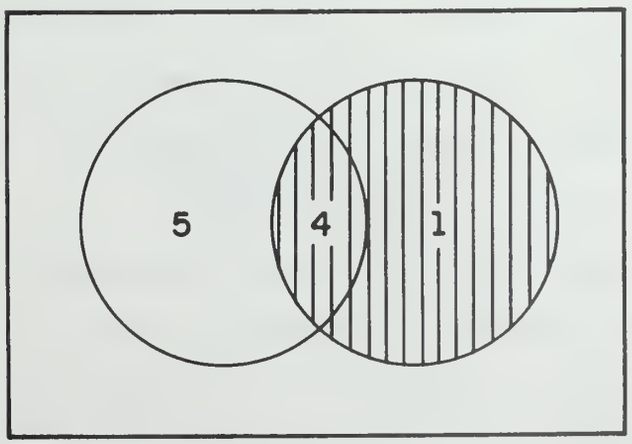


Figure IV.6-e

Equivalent of figure IV.6-c for case 1A.

Case 1B. Assuming that $F_1 \cap C_o^F \neq \emptyset \neq F_1 \cap C_1^F$.

In general it is somewhat unrealistic to expect that we will always get a pair of updates u_o and u_1 as before (case 1) in which u_1 does not modify fields in C_o^F or C_1^F but u_o does. This is especially true because we haven't established any restrictions on the types and order of the updates. It is much more logical to expect that in general updates either do or do not modify fields which are used to test applicability of an update. The latter is case 1A. The former will be covered now, but our discussion of case 1 will certainly simplify our explanation.

For this case figures IV.6-b and IV.6-c become figures IV.6-f and IV.6-g. We have three new regions. As was expected, we now have some new considerations; i.e., the application of u_1 before u_o can: a) expand the domain of u_o by changing some fields in some records in such a way that they will satisfy C_o even though they did not before u_1 was applied (region 8), and b) diminish the domain of u_o by changing some records in such a way that they will no longer satisfy C_o. (Region 6 contains all records which should be subject to both updates; region 7 contains all records which should be affected only by update u_o. For these two regions the application of u_1 before u_o has caused the corresponding records to not satisfy C_o at τ.)

As the reader can clearly observe the situation has gotten fairly complex. In case 1 some work was saved by not having to touch all the records which satisfied $\overline{C}_o \cap C_1$ (i.e. Region 1). In a Johnson's-type model further simplification exists since rarely will two consecutive updates conflict. This is because C_i refers to a single elements address; i.e., two updates are in conflict (in $C_o \cap C_1$) if and only if they refer to the same record. Unfortunately nothing similar can be said here. Condition $\overline{C}_o \cap C_1$ is now satisfied by records

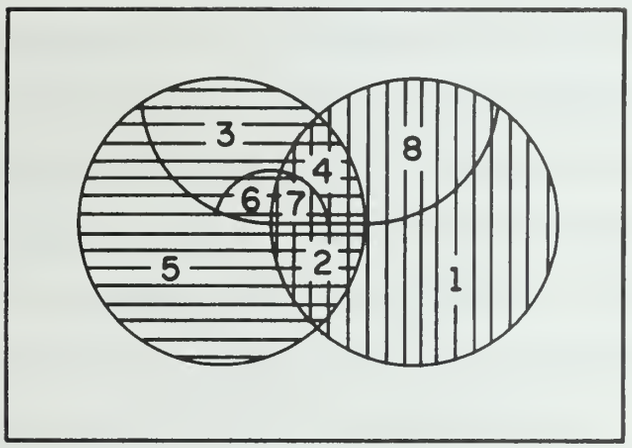


Figure IV.6-f

Equivalent of figure IV.6-b for case 1B.

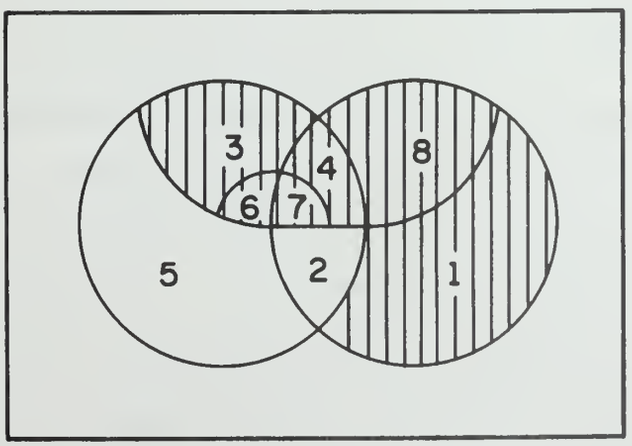


Figure IV.6-g

Equivalent of figure IV.6-c for case 1B.

in regions 1, 6 and 7 and some undoing will in general be required to detect this situation (e.g., for assignments).

Case 2. We have been dealing with the case of two updates. We'll now discuss the case of more than two updates. Suppose that a series of updates u_1, u_2, ..., u_n have been applied to the data base when our conflicting update u_o arrives. There are n! ways in which the n unordered updates could be organized. However, our protocol establishes that the application of an update implies that it is manipulated in such a way that it will be equivalent to having gotten it in the right order. Therefore we can assume that the n updates have arrived and have been applied in the right order (order = u_1, u_2, ..., u_n, with tags obeying $t_1<t_2<t_3<\ldots<t_n$). Thus the observed arrival order is $u_1u_2\ldots u_nu_o$ ($t_o<t_1$) and u_o should be transformed to a set of updates u'_{oj} in such a way that the data base will end up in the same situation that it would have if the right order (i.e. $u_ou_1\ldots u_n$) had been observed.

It would certainly be very nice if in general we could propagate down the u_o; i.e., transform $u_1u_2\ldots u_{n-2}u_{n-1}u_nu_o$ to $u_1u_2\ldots u_{n-2}u_{n-1}u_ou_n$ and then to $u_1u_2\ldots u_{n-2}u_ou_{n-1}u_n$, etc. by repeatedly applying an algorithm for one of our previous cases. To our distress this is not the case. For example, let's assume that n=2, u_o adds 5 to a field F which originally was set to 10, u_1 multiplies F by 2, and u_2 multiplies F by 3. The correct answer for $u_ou_1u_2$ would be 90 (i.e. ((10+5)2)3). When u_o arrives the value of F is 60; if we transform it to $u_1u_ou_2$ we get 75; and if we attempt to transform this to $u_ou_1u_2$ by just looking at u_o and u_1 (i.e. we undo u_1 by dividing 75 by 2, add 5, then multiply by 2) we obtain a value of 85 for F. The interaction between the different updates is also clear when we extend case 1B to cover the multiple update case.

Extending figures IV.6-f and IV.6-g to cover this case is impossible in a two dimensional plane. The u'_{oj}'s specifications also get very complex. The only easy and logical way out is to undo and then redo out-of-order updates.

There is just one singular case which could easily be dealt with. If in case 1A we require our updates not to have any interaction with each other (i.e. homing operations) we could easily establish for which records we should neglect u_o. That is, u_o should only be applied to all records which satisfy $C_o \cap \bar{C}_1 \cap \bar{C}_2 \cap \bar{C}_3 \ldots \cap \bar{C}_n$, since any application of u_o to a record which satisfies any other condition $C_i (i>0)$ will clearly be superseded by u_i. We have ended up with a version of Johnson's model.

IV.6.2.2 Summary

We have analyzed the possibility of expanding non-primary models by increasing the types of operations that they can perform. However, we have found that in general if we allow any kind of operations which modify the fields which other updates use to select their domains then we are forced to solve all order conflicts by a costly undoing-redoing mechanism. If we eliminate the possibility of updates affecting fields which other updates use to establish their domains, then things get simpler for those records which should be touched by only one update (i.e. we don't need to touch a record to correct the order of an update that does not refer to it). However, some undoing might be necessary for records affected by multiple updates. One exception to the last sentence will occur when we have only homing operations, as in Johnson's model.

In general undoing is not a cheap solution and thus we should consider minimizing its effects. Pipeline delays (see section V.2) are an interesting alternative to minimize the undoing-redoing business. However, they are not

effective protection against the less likely but much more costly cases; i.e. updates that arrive very late and thus produce a lot of undoing and redoing (e.g. when network partitioning occurs, when a host goes down being the sole owner of an update, etc.).

IV.7 Conclusion

In this chapter we have made an effort to compare the three distributed data base models under different situations (one at a time, in a similar least-common-denominator environment, and in general). Our conclusions are summarized in table IV.7-1. In this table we have used a three-letter ranking scheme. "A" indicates the model which is best with respect to the issue indicated. "B" indicates an average performance, and "C" a relatively bad performance. The last column of this table indicates the sections in this chapter where the particular issues were studied.

Aided by table IV.7-1, we may state our conclusions easily. The two non-primary models behave similarly in general. When the clocks in Johnson's model are guaranteed to be relatively well synchronized, we might be better off with Johnson's model; but if this is not the case we should seriously consider switching to the Reservation Center model or adding some automatic clock synchronization as discussed later (section V.1). The major outcome of analyzing table IV.7-1 is that there is a drastic difference in the performance of primary vs non-primary models. Basically, whenever we find an A score for a non-primary model we get a C for a primary model and vice versa. Non-primary models are faster and less sensitive to failures, but much less powerful and with a high memory consumption. If we were asked to choose the best model, we simply could not give a single word answer. Our answer would have to be: if your application is so restricted that a non-primary model is useful and you don't care about memory, then use one of the non-primary models. Which of those you should

Table IV.7-1

Comparison Summary

	Johnson's	Reservation Center	Bunch's	See Section
How well do they synchronize (i.e. maintain the "real" update order)	C	B	A	4,5
Local application delay	A	A	C	5
Non-local application delay	A	A	C	5
System's throughput	A	A	C	3,5
Sensitivity to single failures	A	B	C	5
Sensitivity to multiple failures	A	B	C	2,3,4,5
Memory consumption	C	C	A	2,3,4
Operations they support (power)	C	C	A	6

A = best B = average C = worst

choose would depend on the issues of clock synchronization and sensitivity to failures. Otherwise, use a primary model with a unique update application order. This last choice tends to lose something (throughput for example), but you end up with a much more general model applicable to many more environments than the other two models. Interestingly enough, all the available literature in the file allocation problem (see section II.2 and appendix A) do not mention any synchronization between the copies. This leads us to assume that most researchers are headed for a Johnson-like model with the corresponding restrictions.

In our next chapter we will cover some of the issues left open in our discussion of the three models. We will present a clock synchronization

protocol aimed at removing the "C" for Johnson's model in the first row of table IV.7-1. We will then evaluate the idea of a delay pipeline, which we mentioned so many times in section IV.6. Clock synchronization and a delay pipeline seem to be the only improvements which can be added to the non-primary models at reasonable cost. The last section of the next chapter will discuss the highly important area of reliability (or resiliency). Because of the results summarized in table IV.7-1 and our comments above, our discussion of resiliency will be in the context of primary models exclusively.

Chapter V. SOME EXTENSIONS TO THE MODELS

As an outcome of our discussion in section IV.7 we are going to present three extensions to the distributed data base models. We will start (section V.1) with an extension to Johnson's model in order to solve the clock setting problem. In section V.2 we will discuss the idea of pipeline delays for non-primary models. Finally, section V.3 will be directed to the study of resiliency in a primary model.

V.1 Clock synchronization for Johnson's model

V.1.1 General discussion

To obtain a software clock synchronizer, some detection mechanism for an unsynchronized situation must be available. Fortunately the inherent characteristics of the clocks themselves provide us with such a mechanism. If the clocks are perfectly synchronized, then all the updates we get should have a time-stamp with a relatively old time. That is, if we get an update at time τ_G, it should have a time-stamp with time τ_T such that $\tau_G > \tau_T$ and $\tau_G - \tau_T = d$, d being the network delay experienced by the update. Conversely, if we get an update with $\tau_T > \tau_G$ we know that the clocks are not synchronized.

Lamport [L1] presents a method of using the above idea to maintain a synchronized environment. He suggests that, upon arrival of an update, the receiving data base manager should reset its clock to the maximum of τ_G (its own clock) and $\tau_T + \mu$, where τ_T is the time indicated in the update's time-stamp and μ is the minimum expected delay for the network transmission of the update. If we add to Lamport's scheme the requirement that any data base manager which starts operations should first obtain a message (update, special log-on message, etc.) with a recent time-stamp (from which it can set its clock), we end up with a fairly reasonable solution.

The only problem that we can see in Lamport's scheme is that, even though he starts with a real physical clock, he is slowly drifting away from the real time in the positive direction. A fast clock would cause the complete set of clocks to behave as fast as itself. This drifting is inconsequential as long as every clock drifts similarly, which is likely in a majority of cases. Unfortunately, uniform drifting does not happen in every possible situation. When a network partitioning occurs, as mentioned in various parts of chapter IV, we could end up with various subnetworks, each following a different drifting pattern.

If partitioning is considered, we find that the idea of moving away from the real physical time is not appealing. Sites in different subpartitions will generally be unsynchronized. The situation will get worse as time advances. Thus, if partitioning is a serious consideration, we should try to keep the clocks as close to the real time as possible. This should cause time differences between subpartitions to be small, even when a new group of sites comes into operation without any previous contact with the network.

The major problem with trying to follow the real time as closely as possible arises when we are faced with having to set a clock backward. If we have to set back the clock of a given data base manager, we are in danger of violating the logical ordering of its updates. Setting a clock back should never bring about the possibility of generating an update with a smaller timestamp than a previous update. This complication of backward resetting is the main reason why Lamport includes only forward resetting in his scheme.

If we analyze the situations in which a clock might need to be set back, we find that a majority of the backward resettings should change the clock by only a small amount. Resetting a clock backward by a large amount should occur infrequently, and generally involve only a single clock. Given the above

characteristics, we present a scheme which will provide us with clock synchronization while maintaining the clock time reasonably close to the real time.

Our scheme is based upon a resynchronization procedure. This procedure is expected to set the clocks within a small margin of each other. After this setting is completed, all data base managers enter into what we call normal operation. All data base managers are expected to be in an alert state during normal operation and react immediately to any clock asynchrony by calling upon the resynchronization procedure to restore order.

The main feature of our scheme is the "window". We say that two clock values, τ_1 and τ_2, are within a window w if $|\tau_1-\tau_2|\leq w$. (See figure V.1-a.) They are within an "open window" w of τ_1 if $\tau_2-\tau_1\leq w$ (see figure V.1-b). The resynchronization procedure will set the clocks within a small window S_w. During normal operation, the data base managers watch that all incoming messages are within a big open window B_w.

During normal operation each manager is expected to be alert to any asynchrony that may appear. In order to do so, every receiving manager will decide whether the time-stamp of an incoming update is inside its big window. If this is the case the update is accepted and a positive acknowledgment is returned; otherwise the update is rejected and a "reject" message returned.

When a data base manager gets a "reject" message, it immediately knows that there is a discrepancy between the two clocks involved; i.e., that his clock is quite far ahead of that of the manager that sent the reject. Thus, when a "reject" message is received, the resynchronization procedure should be called upon.

The resynchronization procedure will be discussed in section V.1.2. In section V.1.3 we will present our final remarks on this scheme while section V.1.4 will be used to sumarize our conclusions.

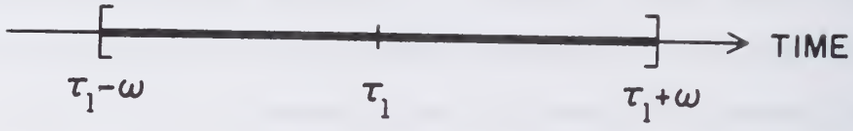


Figure V.1-a

The thick line indicates all the possible values that τ_2 could take on to be within a window w of τ_1.

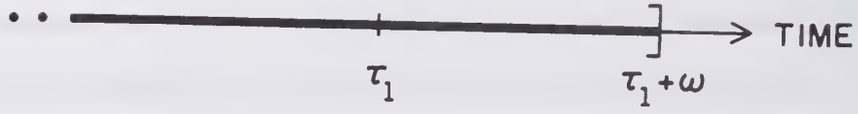


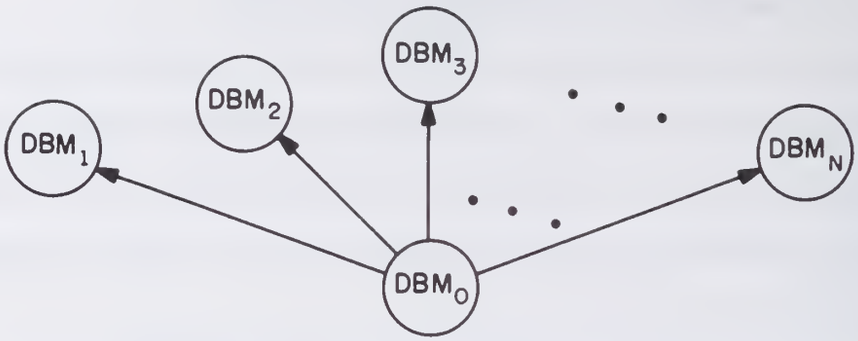
Figure V.1-b

The thick line indicates all the possible values that τ_2 could take on to be within an open window w of τ_1.

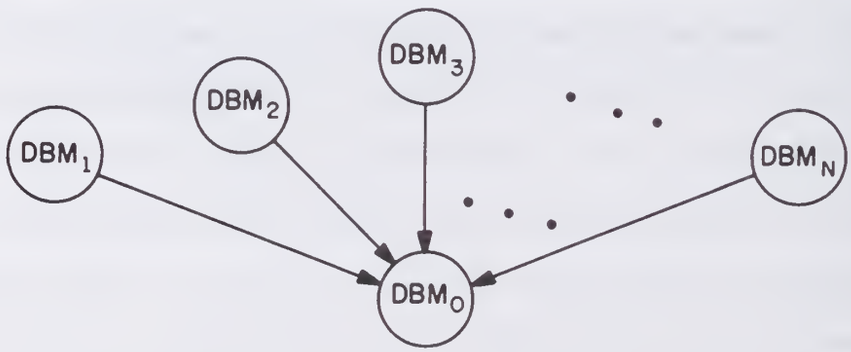
V.1.2 An algorithm for synchronizing the clocks

As was mentioned before, the goal of the resynchronization procedure is to put all the clocks within a small margin (optimally zero) of each other. Operationally, the goal is to synchronize all the clocks to be within the small window of the synchronizing data base manager. The procedure proposed to obtain this goal follows (see figure V.1-c):

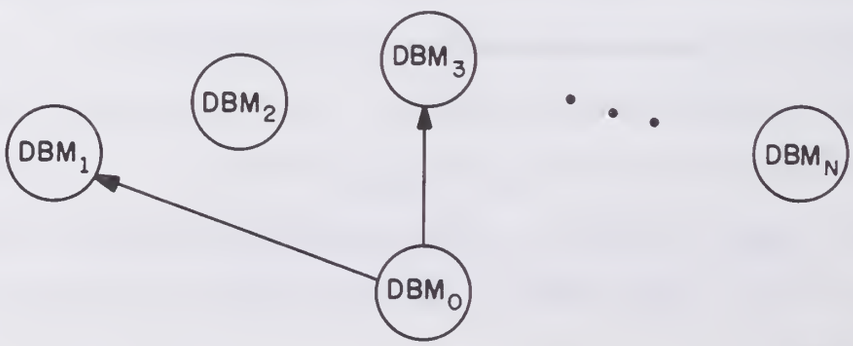
1. The synchronizing manager will send a "what time is it?" message to all the data base managers and note the time at which the message was sent. (See figure V.1-c,i.)
2. Every data base manager will answer this message as quickly as possible with a "my time is xxx" message. The synchronizing manager will note the time at which these messages return. (See figure V.1-c,ii.)
3. The synchronizing manager will estimate the actual time of another manager by assuming that the network delay D for the round trip (both messages) was spent equally in both directions; i.e. that the difference between the clocks is the difference between the synchronizer clock at the time when the "my time is xxx" message arrived and the estimated clock value of the second data base manager obtained by adding the value in this message (the xxx) to D/2.
4. Once the clock differences for all other working data base managers are established the median value M is obtained. In case there are an even number of clocks, its own clock difference (zero) could be used to obtain a single median. (Thus the smaller of the two middle values is chosen.)
5. All computed differences are checked to see if they are inside the small window of M. If this is the case, the clock of the synchronizer is adjusted by M and the procedure terminates.



i) The synchronizer (DBM_0) sends a "what time is it?" message to all other managers (step 1).



ii) All managers eventually return the "my time is xxx" message (step 2).



iii) After proper evaluation of the clocks (steps 3-6) the synchronizer sends "adjust" messages to the managers which require adjustment (step 7).

Figure V.1-c

Illustration of the steps taken by the resynchronization procedure

6. If not all the values are inside the small window, the values outside are evaluated to determine whether their discrepancy could be due to an unreasonably long network delay or is actually caused by clock discrepancy.

To assist us with this evaluation, a table of expected network delays between the data base managers could be precomputed. New estimates of the clock differences can be obtained by assuming that the network delay (on either the outward or return trip) might have been the expected one. If such an adjustment could put the desired difference inside the small window, one or two additional "what time is it" messages could be sent and step 5 repeated. (For example, suppose that S_w is 3 seconds, D is 4 seconds, the expected one-way delay is 1 second, and the difference between the stated clock times is 5.5 seconds, the remote site being behind. The synchronizer computes an estimated time difference of 5.5-2=3.5 seconds, which is outside the window. But from the expected delay (1 second), it notes that the round trip was too slow and there might have been an extra delay in the return. It therefore tries once or twice more to see if D can be brought down to a more reasonable figure. A more sophisticated, statistical approach might be desirable.)

7. If the evaluation of step 6 shows that there is no doubt that there exists a significant asynchrony, or after several tries of the "what time is it?" message we still find that the small window criterion is violated, then an asynchrony is declared. An "adjust" message is then sent to the violating clock(s). The "adjust" message includes a signed value which must be added to the clock to bring it to the median value. A data base manager might have to send an adjust message to itself. (See figure V.1-c,iii.)

After the above steps are concluded, the synchronizing manager becomes an ordinary, working, data base manager and starts acting as such.

In step 4 the median was used instead of the mean value because this measure is less sensitive to errors in the settings of a few clocks. For example, if out of 60 clocks all but one have the same correct time and the remaining one is wrong by 20 hours, the mean value would indicate that the estimated time should be about 20 minutes away from the right time. Moreover, all applications of step 6 that successfully make the estimated value of the clock lie inside the small window after a second or third try would surely affect the mean value, but probably would not modify the median (unless the network delays have a large variance, a possibility that requires some study but seems unlikely).

More than one data base manager might decide to resynchronize simultaneously. If we let them all succeed, a double (or multiple) adjust message could be sent to a given data base manager in order to correct the same discrepancy. This would have obvious undesired consequences. To solve this problem, we could allow an additional valid answer to the "what time is it?" message. If manager DBM_1, after undertaking the resynchronizing process, gets a "what time is it?" message (from manager DBM_2) while in steps 1 to 3, it will decide whether manager DBM_1 or DBM_2 has priority according to a pre-established rule of precedence. If DBM_1 has precedence it will respond with a "cool it" message instead of the normal (for working data base managers) "my time is xxx" message. Manager DBM_1 could also take note of DBM_2's i.d. and notify him when he is done with a "your turn" message. (This is clearly optional.) If, on the other hand, DBM_2 has precedence, then the answer should be a "waiting for you" message and all synchronizing activity should be suspended until a "your turn" message arrives (or possibly a given time-out expires). At this moment the resynchronization procedure will be restarted in step 1.

If DBM_1 is already past step 3 (steps 4, 5 or 6 could be used instead, but using step 3 guarantees less time lost), then it has precedence and should send a "cool it" message. Note that only one manager can be in this stage, since step 2 involves a delay until all answers to the "what time is it?" message return, and a "cool it" message inhibits any further activity.

Step 7 includes the notion of an "adjust" message. Some care should be given to its implementation in light of the problem, discussed before, of setting back clocks. Section V.1.3 will discuss this matter further.

V.1.3 General remarks

The overall behavior of our scheme is then as follows. Whenever a data base manager enters the distributed data base system[1] it will call the resynchronization procedure to set its clock. Once its clock is set it begins normal operating activity. If during normal activity a data base manager detects a violation of its big window it will reject the corresponding update. This rejection will cause the violating manager to perform a resynchronization.

As an outcome of a resynchronization, various adjust messages could be issued. A data base receiving an adjust message is expected to modify its clock by the quantity indicated. If the adjustment is in the positive direction, then the manager should do it without any hesitation. However, if the adjustment is negative much care should be taken not to violate the local update ordering.

As we mentioned before, clocks would probably only be set back by small amounts of time. In this case the managers could enter an alternative temporary clocking mechanism. This mechanism consists of setting the clock to

---

[1] Later in this section we will introduce a condition to determine when a data base manager enters the distributed system.

the value of the time-stamp of the last update generated and then stopping the clock. The clock is not advanced until the adjustment has been honored (i.e., the time as determined by the synchronizer catches up with the setting) or a new local update arrives. In the latter case the clock will be advanced one time-instant to provide a new value for the update's time-stamp.[1] Care should be taken that the update arrival rate is less than one per time-instant.[2] This ensures that we will eventually end this temporary adjustment status. Furthermore, the manager should apply these updates locally but should not broadcast them until the adjustment is completely honored. This is to prevent the generation of multiple reject messages.

If the adjust message indicates that a data base manager should set his clock back by a large amount, the chances are that no other manager has accepted any of its recent updates. In this case the manager could reassign all the local tags very easily (without violating local ordering). By so doing it will actually synchronize them with respect to the updates of the other managers

It is also possible that a large setback is needed and more than one manager needs such a setback. In this circumstance, we must either recall the updates that were sent to the other out-of-step managers or wait a long period, as suggested for small setbacks. (There may be other equally unpleasant solutions.) Neither of these alternatives seems attractive. The likelihood of having a large setback for multiple managers is small. (We will show how to make it even smaller in the next paragraph.) We will then not feel guilty in suggesting that the long waiting period should be adopted.

---

[1] Note that this is very likely the time-stamp that the update would eventually get if we stopped activity until the clock catches up.

[2] Note that if updates arrive faster than the clock ticks, we are in trouble anyway, since there is not a unique time-stamp to put on each update.

As mentioned before, it is not very likely that there will be several clocks with a similar wrong setting. However, one single wrong setting could possibly propagate to the full distributed system. For example, suppose that we start with a data base manager with a wrong clock and add managers in such a way that the wrong clock setting always prevails in the resynchronization procedure. Under these circumstances it might be advisable to ask for a quorum before distributed operations are initiated. Thus, a data base manager that comes up and finds that this quorum is not met will simply start working for itself, without broadcasting any updates. As soon as the quorum is met, it could take part in resynchronization[1] and enter into normal operation.

V.1.4 Summary

In this section we have tried to present a solution to the clock syncrhonization problem in Johnson's model (sections IV.2 and IV.7). Unfortunately, trying to synchronize the clocks gets rid of the property of complete clock independence which caused the nice behavior of Johnson's model during partitions (sections IV.5 and IV.7).

Clock synchronization could be obtained by Lamport's scheme [L1]. And in environments where partitioning is not a problem this is an appealing solution.

One of the disadvantages of Lamport's scheme is that it is susceptible to clock drifting in the positive direction. This susceptibility makes it unattractive if partitioning can occur.

We introduced a second scheme. This scheme should solve the drifting problem, but the problems of setting a clock back had to be addressed. We had to pay the price of a considerably longer protocol.

---

[1] At this moment, detection of a single wrong clock setting would be very easy. Recovery could be accomplished by reassigning time-stamps as described earlier.

In general we feel that the clock synchronization problem is solvable. The two schemes presented are good examples of how it can be solved.

V.2 Delay pipeline

In various places in chapter IV we mentioned a delay pipeline. By a delay pipeline, we mean that as updates arrive they are introduced at one side of a pipeline. After a pre-established delay D they go out through the other side and are ready to be applied. When an update u is ready to be applied, the delay pipeline is checked to see if there exist other updates with tags smaller than that of u. If there are, we take those updates out of the pipeline and apply them in the appropriate order, following them by the application of u. We are concerned with the probability of having to undo an update (for different values of D), if a precise application order is required. This measure is very relevant for Johnson's and the Reservation Center models for a system such as S6 (section IV-6), for which there is a potential need for undoing. We will start by analyzing delay pipelines for Johnson's model.

In Johnson's model there are two sources that could cause the updates to get out of order: clock asynchrony and network delays. For our discussion we will assume that the clock-time has a normal distribution (which seems reasonable) with mean c and standard deviation σ_c. However, we do not know enough abou network delay. For the ARPA network Naylor et al. [N1] and other researchers (Kleinrock [K1]) give delays only as a network mean value or as mean values for different numbers of hops. The means are around 50 milliseconds for 1 hop and 800 for thirteen. In order to simplify the approach (since we are only concerned with gaining some insight into the problem), we will assume (obviously wrongly) that the network delay behaves like a normal distribution with mean n_d and standard deviation σ_d. The parameter n_d will be given values 10, 100 and 1000 milliseconds, with the hope that the actual behavior lies somewhere near the results

obtained. There is a problem with estimating σ_d, since there is no data. Our solution will be to assume a large spread. This will be done with care since we do not want the results to be invalidated by the effect of including meaningless negative delays. We will then make $n_d-5\sigma_d$ correspond to zero delay.

Let's now take a data base manager and assume that his clock is set to time τ_1. We want to evaluate the probability that at time τ_1+D all updates carrying a tag smaller than τ_1 have arrived. This is the same as the probability $P(\tau_1,D)$ that all updates generated up through local time τ_1 have arrived at the given data base manager by time τ_1+D. The distribution of arrival times of updates with tags = τ_1 is normal with mean $n_d+\tau_1$ and standard deviation $\sqrt{\sigma_d^2+\sigma_c^2}$ [H1]. In figure V.2-a we show the upper half ($P\geq.5$) of the $P(\tau_1,D)$ curve for various network delay parameters. As expected the probability of our data base manager's having received all updates with tag smaller than a given reference time σ_1 increases with time. As shown in the graph, the probability of getting all the updates with tag smaller than σ_1 is .5 at time σ_1+n_d and practically 1 at time σ_1+6 seconds (for the 3 graphs of figure V.2-a). This implies that the inclusion of a 6-second delay pipeline will very nearly solve the problem. Equivalent statements with smaller D could be obtained from the graph.

Obtaining results for the Reservation Center model is a very complex problem. In the description of this model (section III.4), we pointed out that the sets of tickets distributed by the reservation center could be any collection of instructions or numbers. Actually, we also pointed out that in some sense Johnson's model is a particular case of the Reservation Center model (with $m=\infty$). It is thus very hard, if not impossible, to say anything about such an undefined situation.

If we restrict ourselves to situations in which we allow only numbers in the sets of tickets issued by the reservation center, then the situation is

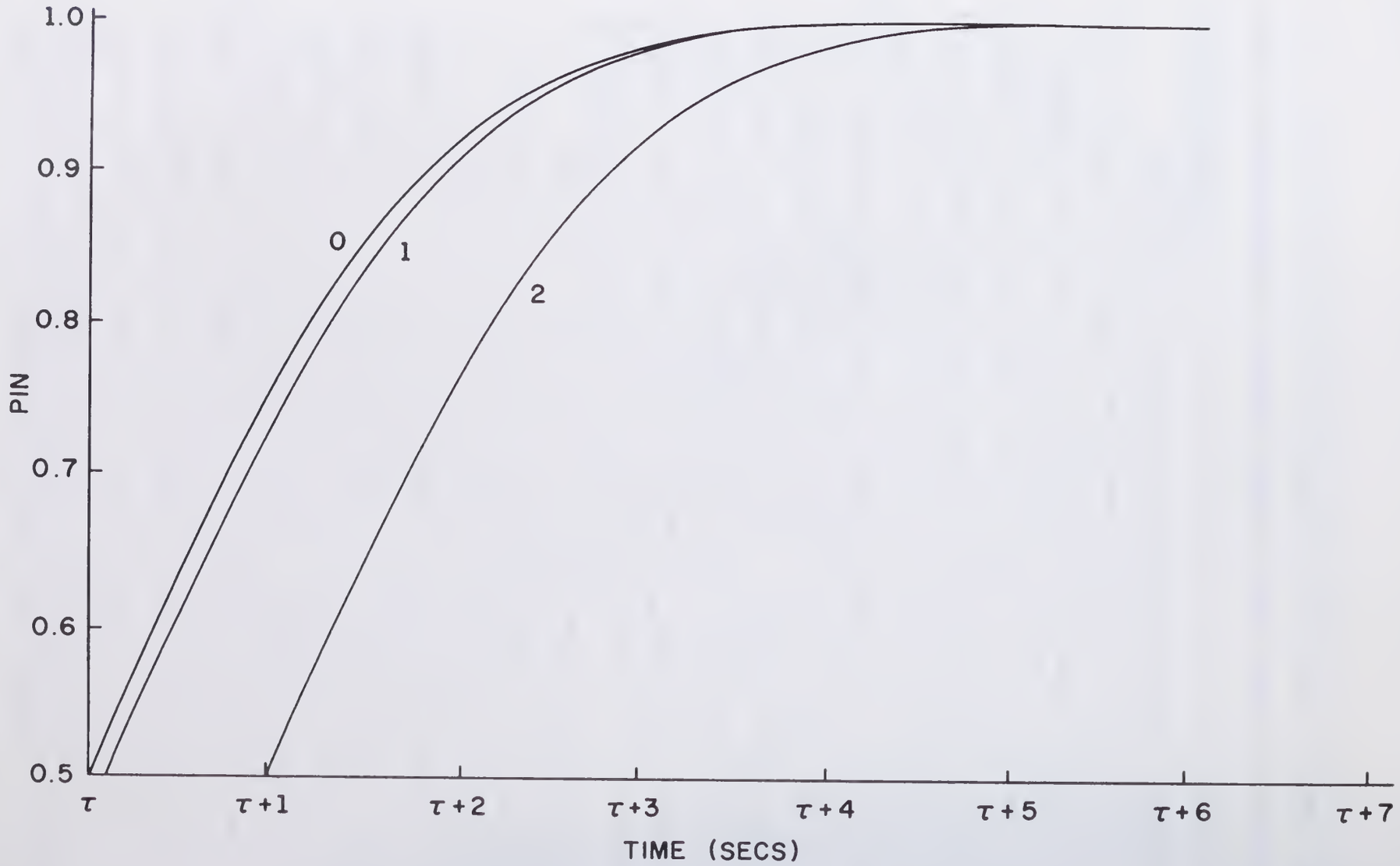


Figure V.2-a

Graph 0 uses N(.01,.002) for network delay, graph 1 uses N(.1,.02) and graph 2 uses N(1,.2). Pin = probability of getting all updates in order if we use a pipeline with delay D - N(a,b) denotes normal distribution with mean a and standard deviation b. Parameters a and b are given in seconds.

somewhat simpler. However, in order to have some specific results we have to know something about how often a data base manager runs out of tickets and starts using the next locally available set of tickets. In general we do not have this information. Thus, rather than give a distribution of the probability P as above, we will settle for an approximation to the value of D required to ensure an almost 100% probability for P.

In an RC(m,n) model we have that each data base manager maintains n sets of tickets locally. In a period of hyperactivity a data base manager DBM_i could then be n-1 sets ahead of the other data base managers. DBM_i might get further ahead if it is in a location such that it gets the sets of tickets before other managers (e.g., very close to the reservation center). This advantage would probably not be more than 1 or 2 sets (if the network delay is relatively short). Thus, it is almost impossible that DBM_i be ahead by more than (n+3) sets as compared to any other manager. If DBM_i maintains this advantage, and the reservation center keeps on sending sets of tickets every time period of length m (a very unlikely situation because the "exhaust" messages should speed up its activity), then it should take on the order of (n+3)m seconds (including a few network delays which were assumed relatively small) for DBM_i to be sure that it won't get any relatively old update (according to its tag). In summary, a value D on the order of (n+3)m will seem to sufficient to ensure that we will rarely have to undo an update.

It has been our intention to discuss the feasibility of using delay pipelines to minimize the probability of having to undo updates in such systems as S6 (section IV.6). Throughout our discussion we have assumed that all sites are operational all the time (i.e. no failures). Under these circumstances it seems reasonable that a delay pipeline will prove a significant advantage. However, this advantage is at the expense of giving up the A rank in table IV.7-1

for the local and non-local application delay. Furthermore, delay pipelines with reasonable D's cannot offer protection for situations in which failures occur (i.e., reality). A computer system that fails while it is the sole holder of an update and then comes up later and tries to broadcast this update will be a real headache because, even if this seldom happens, it would require a tremendous amount of undoing.[1] A similar but more severe situation would arise when network partition occurs.

V.3 Resiliency

V.3.1 Introduction

As we pointed out at the beginning, our major goal was to obtain some guidelines for the design of a workable distributed data base system. We have studied the three available models and come out with the conclusion that non-primary models lack power (section IV.6). Increasing their power is rather difficult if not prohibitively expensive. On the other hand the relative independence of the data base managers (especially Johnson's, see section III.2, IV.2 and IV.5) makes the reliability of these models easily obtainable. (Basically it suffices to provide a straightforward protocol to prevent lost messages.) In general we would expect a majority of applications to be either willing or forced to use the much more powerful approach of a primary model, such as Bunch's. We are thus clearly concerned with upgrading this model to a workable state by eliminating the problems presented in section IV.3. In summary, we still have to present the guidelines of a workable model. In order to satisfy our goal, we will do so by presenting some models based on the primary-backups ideas introduced by Bunch.

---

[1] Since the system is the sole owner of the update, reassigning the tag might be the right thing to do.

In an effort parallel to this thesis we have participated in work on multi-copy resiliency techniques. Most of the concepts that we are going to present here were already presented in a previous report [A4].

In any network it is always possible to have a very high number of simultaneous failures. Once we accept this simple fact, it is clear that we are incapable of designing an error-free system. Any detection mechanism that we know of could be overwhelmed by the proper sequence of errors. Thus, given that perfection is unattainable, we generally settle for minimizing the probability of an undesired failure within the constraints imposed by our budget and other requirements. We thus introduce the concept of n-host resiliency.

By n-host resiliency we mean that at least n-hosts must be aware of an update before it can be either acknowledged to the user or applied at any local data base. Thus, in order for an update loss to threaten the distributed system's reliability, it is necessary that n or more hosts fail "almost simultaneously." It should be noticed that after one or more (but less than n) host failures occur, the distributed system will keep on working (if n or more hosts are still available) and continue the transmission of any given update until it reaches all available hosts. After a single or multiple host failure occurs, we could have a situation in which fewer than n hosts have a message. However, continued transmission of the update to further backups will eventually cause n or more working hosts to have the update. At this moment the "almost simultaneous" period ends, and the system would be capable (if enough hosts are available) of recuperating from another set of less than n "almost simultaneous" host failures.

The n-host resilient models that we are going to present are able to solve most of the problems discussed for Bunch's model (e.g., missing updates; see sections III.3 and IV.3); but the network partitioning problem is still with us.

Network partitioning is mainly a function of the network topology. Requiring all pairs of sites to have a communication channel between them (i.e., a fully connected network) would increase the reliability to such a point that we could probably ignore the partitioning problem.[1] This is an extreme and probably very expensive solution. In general we cannot allow the subnetworks to maintain their normal activity, because the data base copies can easily become irreconcilable. For a majority of applications we could probably get along by providing a degraded service. An example of such service is our palliative solution of section IV.4 in which only the subnetwork with a majority (possibly weighted) of the sites keeps on working and all others restrict themselves only to queries. In the following discussion we will assume that the partitioning problem has been solved (e.g., by full connection, our palliative solution, or some other technique) and restrict ourselves to the other aspects of the resiliency problem.

Two models will be discussed next. We will first (section V.3.2) present a broadcast model, which we have developed, in some detail. Then (section V.3.3) we will briefly present the main features of a chained model, developed by Alsberg and Day. Both are discussed in [A4] but some minor changes have been made in the broadcasting model. Finally, in section V.3.4 we will discuss the advantages, with respect to reliability, of such models.

The main difference between the two models that we are going to describe is the way the updates are sent to the backup copies. In the broadcast model the primary will send the updates to all the backups simultaneously, while in the chained model the primary sends its updates to only one backup, which in

---

[1] In such a network at least m-1 (where m is the number of sites) simultaneous failures are required for a partition to occur.

turn sends it to the next one, and so on. This one-at-a-time transmission always follows a pre-established order.

V.3.2 The broadcast model

To simplify our discussion we will describe 2-host resiliency unless otherwise noted. We should point out that generalizing to a higher-order resiliency scheme is expected to be a straightforward matter.

In the broadcast model we assume that we divide the network hosts into three groups: users, primary and backups. For simplicity we will assume that these groups are disjoint. Relaxing this assumption and allowing the primary or the backups to be users is a straightforward matter, which in some ways allows some simplifications. The primary and backups together form the set of "server hosts."

A user host is a site which is capable of receiving queries and updates from the external world. The primary and backups are the only hosts with a copy of the data base. An explicit linear ordering of these hosts is assumed, with the primary as the first node and the backups following any arbitrary pre-established order.

Communication with the user host takes place as follows. The user host sends its updates or queries to any server host. The receiving host will try to answer all queries itself. However, all updates will be sent to the primary. In some cases, the site receiving the request from the user is the primary.

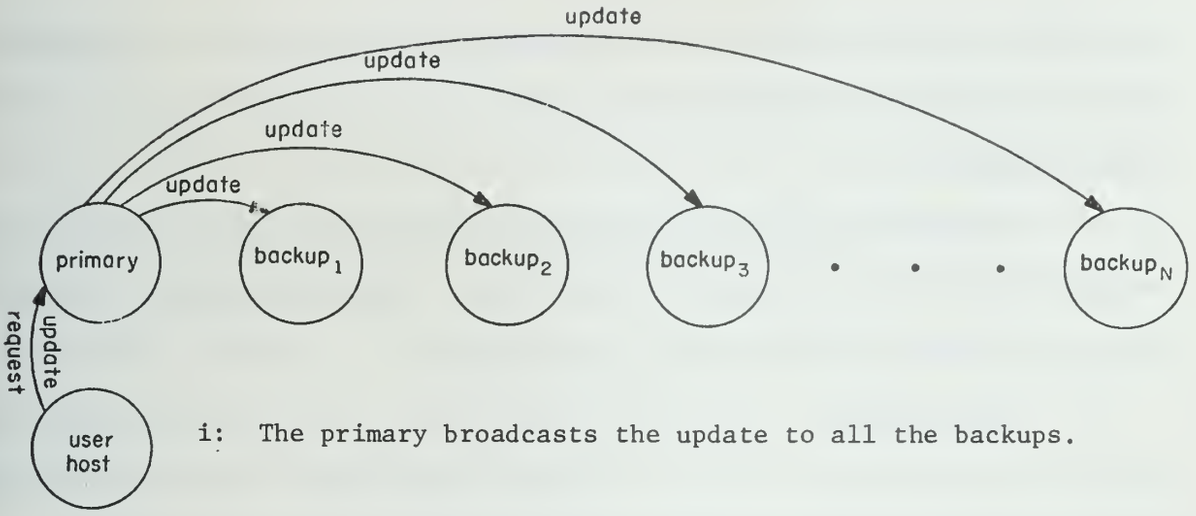
In any case, the primary ends up with the update. Then a second stage of the update is initiated: its actual broadcasting to all data base copies.

Update broadcasting starts with the primary assigning a sequence number to the update and journalizing it locally. The primary then will broadcast the update (with the sequence number attached) to all existing backups.

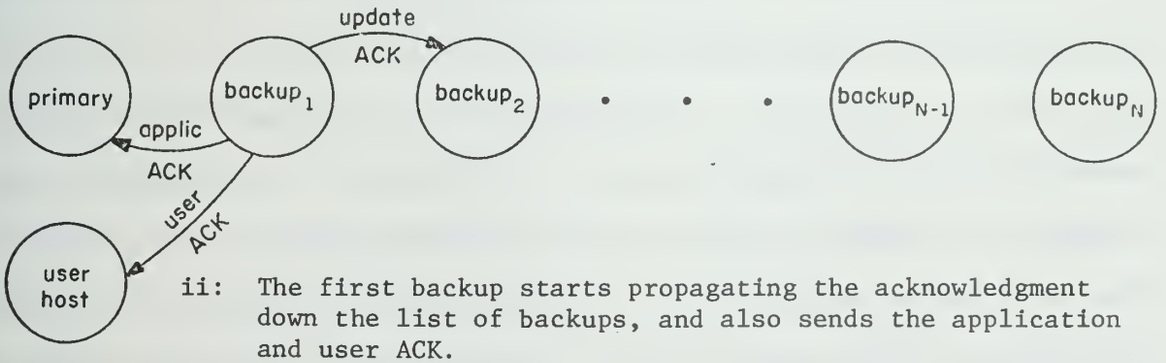
(See figure V.3-a-i.) When a backup receives an update it will verify that the update's number is sequentially correct before considering it for local application. That is, the backup checks for missing updates.

Upon reception of an update, the first backup in the linear order is expected to send an acknowledgment (which includes the update's number) to the host that follows it in the given linear order. (See figure V.3-a-ii.) All other backups wait passively until the acknowledgment gets to them. The acknowledgment includes a counter, the resiliency counter. This counter is going to reflect the number of hosts which are known to have seen the update. Thus, the initial value of the resiliency counter is always 2; i.e., the primary and the first backup are known to have seen the update. At the time that any backup gets an update acknowledgment it will simply pass the acknowledgment to the next backup in the linear order after adding 1 to the resiliency counter. The last backup sends an acknowledgment to the primary. (See figure V.3-a-iii.) This last acknowledgment indicates that the update has been received at all available backups and the second stage of the update application is complete.

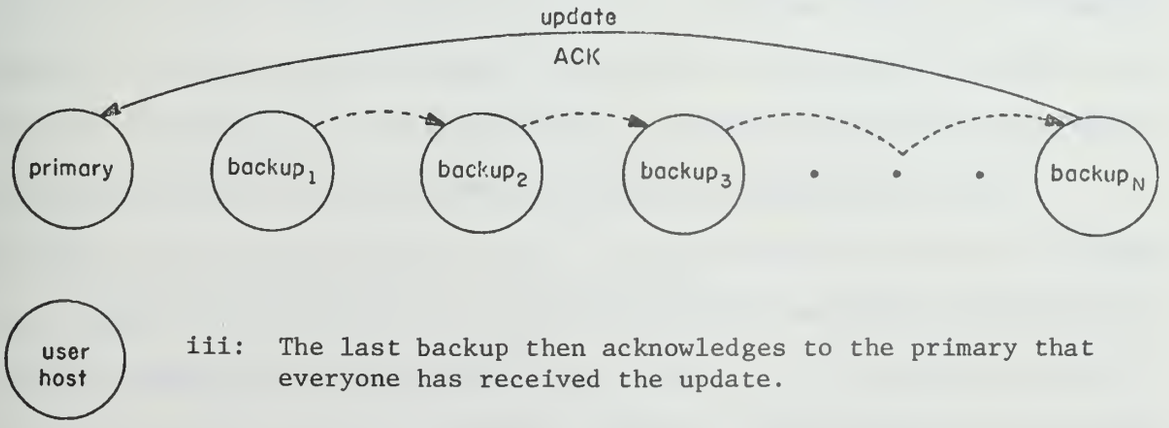
The third and last stage of the update is its actual application at all data base copies. Given the requirements of 2-host resiliency, no host should apply any update until at least 2 hosts are aware of the update. We thus have to wait until the first 2 hosts in the linear list have the update before actual application can be started. When a backup finds that the resiliency counter has a value of 2 after it has set or incremented it as required by the second stage, it will acknowledge to the user (user ACK) and simultaneously to all the preceding hosts (application ACK) the satisfaction of the 2-host resiliency criterion. (See figure V.3-a-ii.) The primary or any backup host which finds itself with an insufficiently small resiliency counter will wait until the application ACK arrives before starting local application. (For 2-host



i: The primary broadcasts the update to all the backups.



ii: The first backup starts propagating the acknowledgment down the list of backups, and also sends the application and user ACK.



iii: The last backup then acknowledges to the primary that everyone has received the update.

Figure V.3-a

The Broadcast Model for 2-Host Resiliency

resiliency, only the primary can find itself in this situation.) All other hosts will start local application as they receive the second stage's acknowledgment.

Up to now we have considered normal (untroubled) operation. Real life requires that we allow for abnormal situations. Complications arise when (i) a message gets lost, or (ii) a host is unavailable.

i) Lost messages. There are various points in our protocol where a lost message can be detected, such as: When an acknowledgment for an unknown message arrives or when message h+1 arrives while message h hasn't. The following protocol should handle these anomalies.

Upon arrival of message h at a backup manager B, B will first check whether all prior messages have successfully been received. If this is not the case, then B may assume that a message has been lost and ask for the retransmission of missing messages (probably by the primary). In the meantime update h is stored in a "waitlist" until the problem can be cleared up. If, on the other hand, all prior messages have been received, then we proceed with the application of update h (as described before) and we are ready for the acknowledgment propagation. If B is the first backup in the linear list, then it initiates this process by sending the acknowledgment shown in figure V.3-a-ii. Otherwise a waiting period (controlled by a timeout (TIMEOUT1)) is started.

When acknowledgment h reaches B, B first checks to see if it corresponds to a known, processed message. If so, B simply adds its own acknowledgment by adding one to the resiliency counter and propagating the acknowledgment to the next host down in the linear list. However, if it represents an acknowledgment for an unknown message, B can assume that message h is lost and ask for the retransmission of that message (from the primary or probably from any of the preceding backups in the linear list if the protocol is such that they have it

journalized). In this case the propagation of the acknowledgment should be suspended, but a flag (called the first-to-acknowledge flag) should be set to indicate to B that it should restart the propagation of the acknolwedgment as soon as message h arrives. A similar action is taken when an acknowledgment for a known but unprocessed message arrives (i.e. one waiting in the waitlist for the intermediate updates). In this case we will also set the first-to-acknowledge flag and suspend the propagation of the acknowledgment until the update can be extracted from the waitlist.

Multiple failures could occur; for example, a message asking for the retransmission of a lost message could be lost. Therefore, a pair of timeouts are also included. Whenever the retransmission of an update is solicited, TIMEOUT2 is set. Most likely the action that should be taken when TIMEOUT2 expires is to repeat the appeal for retransmission.

As mentioned before, TIMEOUT1 is set to wait for the acknowledgment of an accepted (correct sequence) message. All but the first backup in the linear order will have a similar TIMEOUT1 mechanism. However, the length of the timout might change, increasing as we move down the list. When a TIMEOUT1 expires, a search for the acknowledgment will be started by sending an "ACK SEARCH" message to the preceding backup in the linear order. Reception of an "ACK SEARCH" message for an unknown update will also help to detect lost messages. Reception of an ACK SEARCH for the known message will produce one of the following responses:

1) the retransmission of the acknowledgment (if the acknowledment was sent and lost in transmission),
2) propagation of the "ACK SEARCH" message in the direction contrary to that of the propagation of acknowledgments, or
3) no action because the corresponding message is either in the waitlist or (was lost and) is already going to be retransmitted.

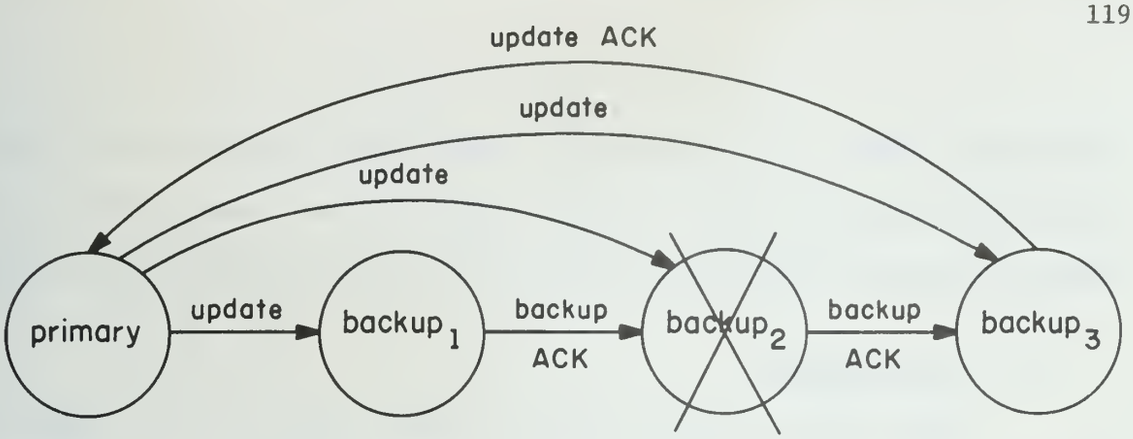
A similar time out mechanism (TIMEOUT3) will be useful to detect lost application acknowledgments.

ii) Unavailable hosts. Partitioning problems were discussed in section V.3.1; they will not be considered here. We will concentrate first on how the system recovers from a single backup failure. Primary failure and multiple failures will be discussed later.

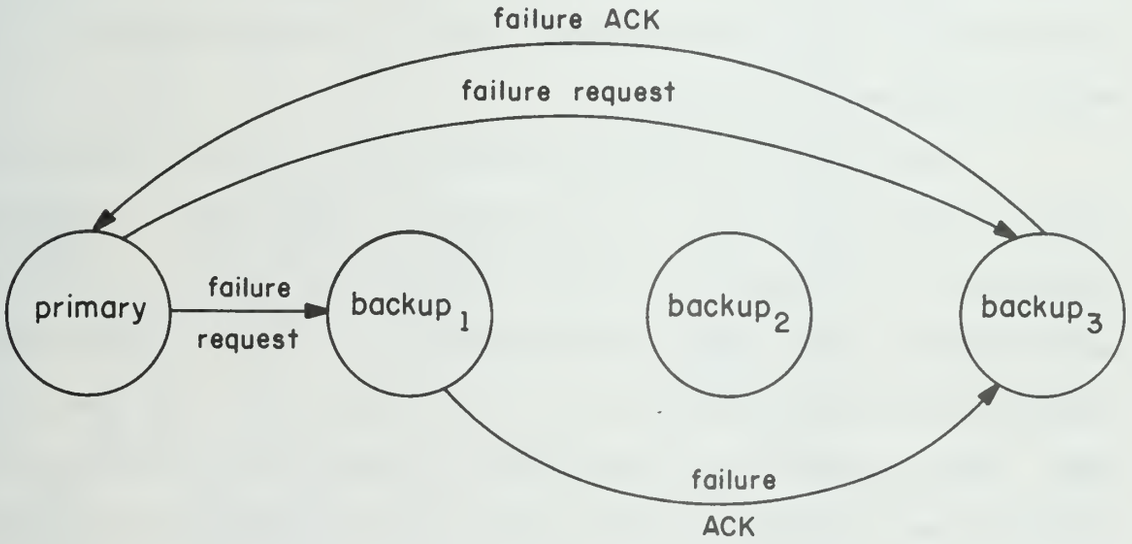
When a backup copy fails, the primary should be notified. In many cases the primary itself will be able to detect such a failure, when it is unable to communicate an update to that backup. (See figure V.3-b-i.) Once the primary has such information it will send a special update, a failure request, using the same protocol used for any other update. When each backup gets this update, it will modify its linearly ordered list. This will cause the two neighbors of the failed backup to bypass it in any future communication. (See figures V.3-b-ii and V.3-b-iii.) As soon as the primary gets the acknowledgment of the failure request message the recovery from the failure will be complete.

When the primary fails, the first available backup in the linear list will be elected to replace it. The new primary must first find out whether it can initiate normal operation. For example, it might have to check whether a majority of the backups is up, whether there are n backups up, etc. If everything is ready for normal operation, the new primary then must find out what the previous primary was doing. If n-host resiliency has been met, there is always a backup that has seen any udpate received by the primary and acknowledged to the user. Thus, either the backup will retransmit the update to the new primary, or the user will get tired (timeout) and retransmit it again. In any case, consistency is maintained and the problem solved.

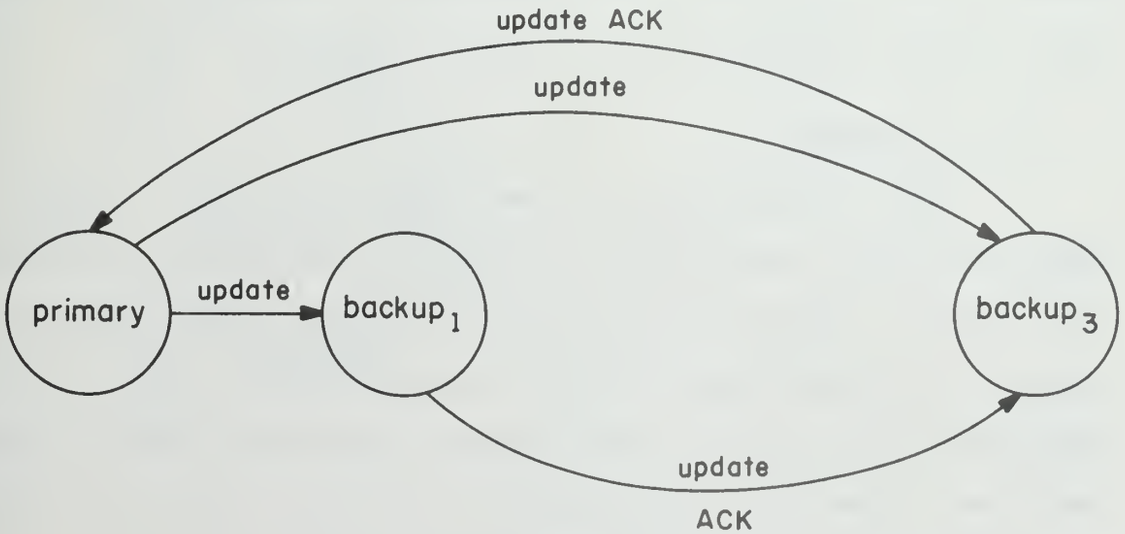
Multiple failures are treated similarly to single failures unless n or more failures occur in an n-host resilient model and the primary is among the



i. A backup Host fails.



ii. A failure message is transmitted.



iii. Communication continues after all communication to failed host is eliminated.

Figure V.3-b. Steps taken to recover from backup host failure

systems that failed. It is possible to detect such a massive failure, and a most likely solution is that everybody should stop any update activity until a sufficient number of the failed systems or the failed primary comes up again.

V.3.3 Other models

Many other variations of Bunch's model (or the Broadcast model) are possible. However, all major points have been illustrated in the previous section (V.3.2). Next we will briefly present the chained model [A4] to show one of such possible variations.

In the chained model the interaction with the user is similar to the action in the broadcast model. The difference is that instead of broadcasting the update it is sent only to the first backup in the linearly ordered list. Any backup that gets an update will acknowledge its reception to the sender (backup ACK) and then retransmit it to the following backup (if there exists one). Upon reception of the acknowledgment (backup ACK) from a backup (or if no following backup exists) the backup will transmit a second acknowledgment (Backup Forward ACK (BF ACK)) to its predecessor. A summary of this message flow is presented in figure V.3-c (similar to figure 3 in [A4]).

V.3.4 Conclusion

With the introduction of n-host resiliency we have clarified the concept of resiliency during multiple failures. We have exchanged the impossible goal of perfect resiliency for the much more reasonable goal of n-host resiliency. However, by requiring that n hosts must cooperate on any action, we open up the possibility that the service may be down for long periods of time, simply because fewer than n server hosts are available. We will next look at this problem for the case n=2. The analysis is largely taken from [A4].

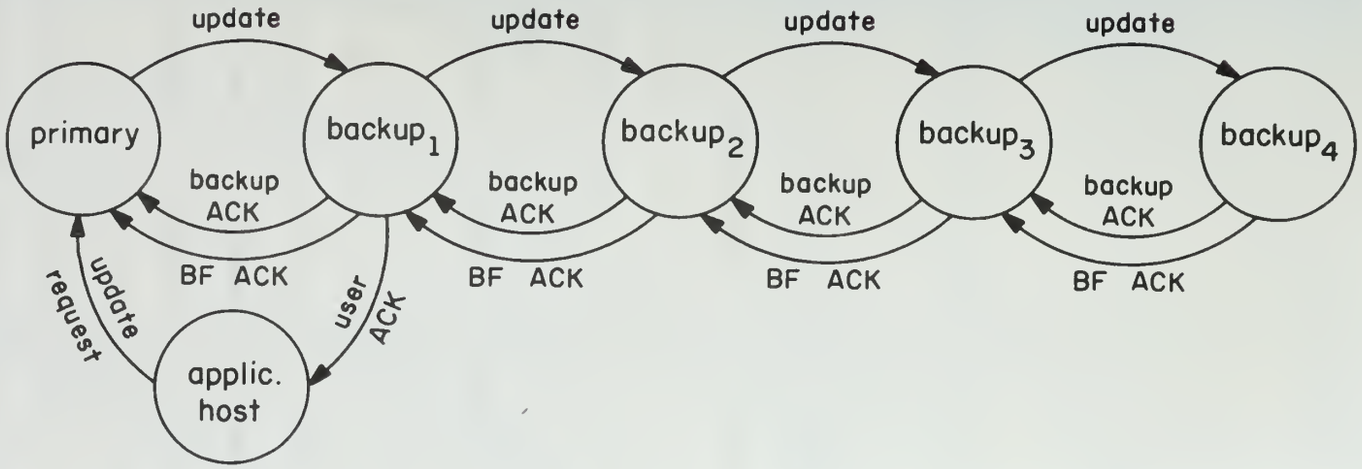


Figure V.3-c

Summary of the message flow for the chained scheme.
(BF ACK refers to the Backup Forwarded ACK mentioned in the text.)

Suppose there are N server hosts altogether. Let A be the availability[1] of any one data base manager. (See section II.3.) Furthermore, assume that failures are independent. The probability that none of the N data base managers is up is

$$P_o = (1 - A)^N.$$

The probability that only one is up is

$$P_1 = NA(1 - A)^{N-1}.$$

Therefore the probability (P) that we can't meet 2-host resiliency is

$$P = P_o + P_1 = (1 - A)^{N-1}(1 - A + NA).$$

We can now define the service down-time per day as P times the proper time constant (i.e., 24 hrs., 1440 min., etc.). The availability A has a more intuitive meaning when presented as down-time per day; i.e., A = (24 - host down-time)/24 where host down-time is given in hours. Using these more intuitive terms we obtain the results plotted in figure V.3-d (figure 7 in [A4]).

More specific results for an average down time of two hours (A = .91667) and one hour (A = .95833) are shown in tables V.3-1 (table 1 in [A4]) and V.3-2 (table 2 in [A4]) respectively. From these results it is clear that, if N is somewhat larger than n, we can safely use an n-host resiliency criterion.

After the presentation of the resiliency models it seems that a study similar to the one in sections IV.5 and IV.6 is in order. If we begin such a study for the broadcasting model we would realize that, with respect to the topics studied, it is almost equivalent to Bunch's model. The main difference between the broadcasting and Bunch's models is in the area of acknowledgment

---

[1] As in section II.3 availability is defined as A = up-time/total-time = up-time/(up-time + down-time).

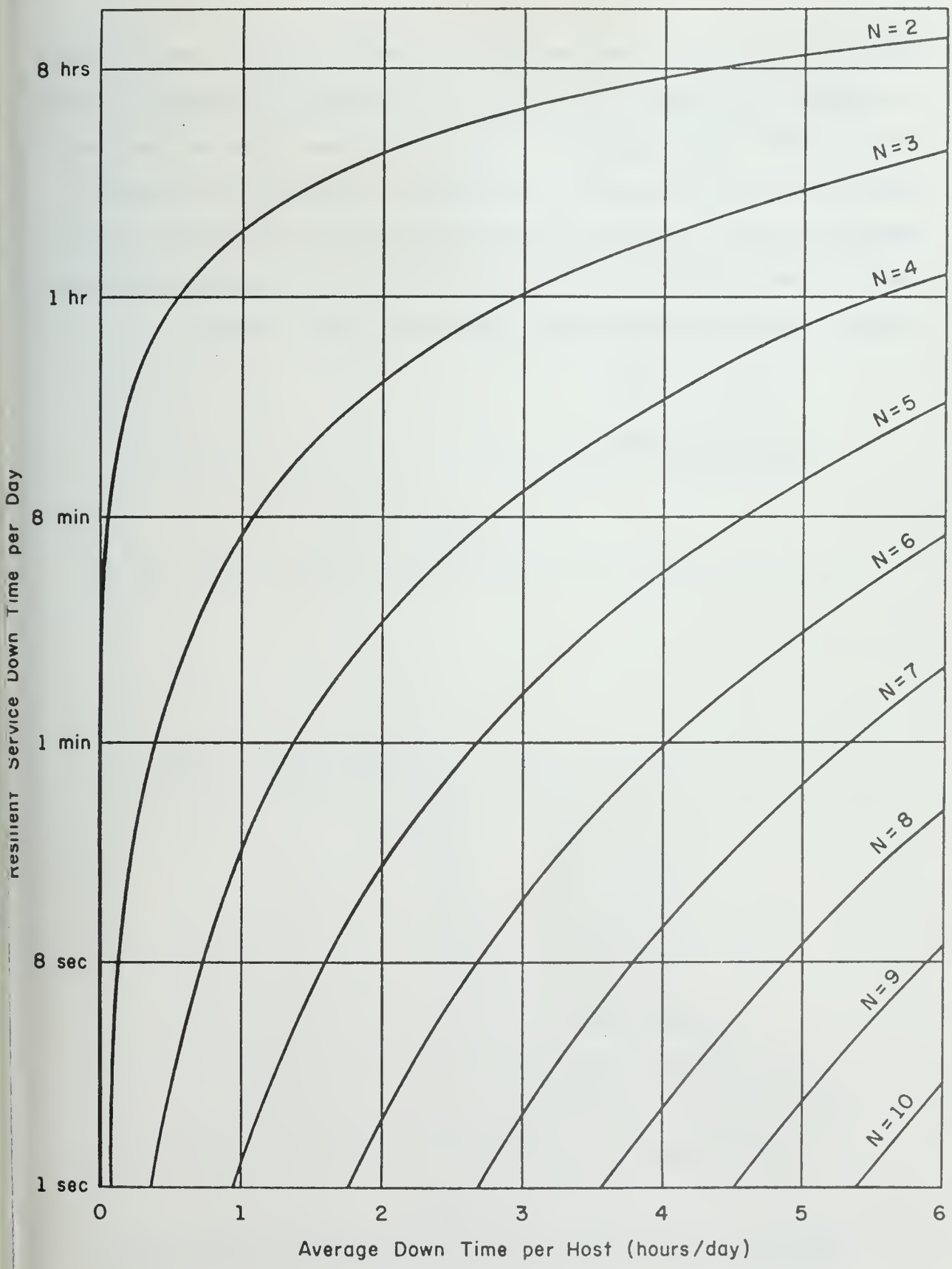


Figure V.3-d. Service Down-Time vs. Host Down Time and Number of Hosts

protocols. But this area was omitted in our previous study because its impact on throughput, application delay, etc., was determined to be negligible. There are a few areas where our chapter IV discussion changes if we use the broadcasting model instead of Bunch's. (An example of such an area is non-local application delay.) But the relative grades of table IV.7-1 are not altered. Similar comments hold for the chained model, although the protocol differences between the chained model and Bunch's model are slightly bigger.

Number of Service Hosts	Service Down Time
2	3.8 hours/day
3	28 minutes/day
4	3 minutes/day
5	19 seconds/day
6	2 seconds/day
7	6 seconds/month *
8	7 seconds/year *
9	1 minute/century *
10	6 seconds/century *

* not plotted on figure V.3-d

Table V.3-1

Service down time when average host down time is two hours per day

Number of Server Hosts	Service Down Time
2	2 hours/day
3	7 minutes/day
4	24 seconds/day
5	1 second/day
6	23 seconds/year *
7	1 second/year *
8	5 seconds/century *
9	2 seconds/thousand years *
10	1 second/ten thousand years *

* not plotted in figure V.3-d

Table V.3-1

Service down time when average host down time is one hour per day

Chapter VI. REVIVAL OF THE FILE ALLOCATION PROBLEM

In previous chapters we have demonstrated the limitations of a Johnson-type (non-primary) model. If we were to write something about a distributed data base in a general sense we would undoubtedly deal with a primary model. Interestingly enough this is not the approach taken by the many researchers in the file allocation area [C1,C2,C3,C4,U1,L2,etc.]. The major concern of these researchers has been to present different techniques to solve the file allocation problem (see section II.2 and appendix A), but they have unfortunately based their presentations on a restricted model of the non-primary type. Luckily their effort is not in any way wasted. A reorientation and reformulation is needed, but the final results should be quite similar.

To argue our point we will analyze Casey's model under a primary-backup strategy.

In Casey's work (see section II.2 and appendix A), the problem is to determine the lowest-cost allocation. The cost $C_1(I)$ of a given allocation I (I = collection of sites that maintain a local copy) may be expressed as:

$$C_1(I) = \sum_{k \in I} \sigma_k$$ storage costs

$$+ \sum_{j=1}^{n} \lambda_j \min_{k \in I} d_{jk}$$ query transmission costs (queries are sent to "closest" data base copy; i.e., the one it is cheapest to transmit to)

$$+ \sum_{k \in I} \sum_{j=1}^{n} \Psi_j d'_{jk}$$ update transmission costs (updates are sent to all copies of the data base)

where:

I = index set of sites with a copy of the file

n = number of sites in the network

Ψ_j = update load originating at site j

λ_j = query load originating at site j

d_{jk} = cost of communication of one query unit from site j to site k

d'_{jk} = cost of communication of one update unit from site j to site k

σ_k = storage cost of file at site k

If we now turn to a primary model, such as Bunch's model, and we neglect acknowledgment traffic (just as Casey did), we end up with a new cost function:

$$C_2(I) = \sum_{k\in I} \sigma_k \qquad \text{storage costs}$$

$$+ \sum_{j=1}^{n} \lambda_j \min_{k\in I} d_{jk} \qquad \text{query transmission costs}$$

$$+ \sum_{j=1}^{n} \Psi_j d'_{j1} \qquad \text{transmission costs to send updates to primary}$$

$$+ \sum_{k\in I-\{1\}} \sum_{j=1}^{n} \Psi_j d'_{1k} \qquad \text{transmission costs to send updates from primary to backups}$$

In this formula site 1 is assumed to be the primary and expression I-{1} refers to the other sites with a copy of the data base (i.e., I-{1} is the set of backups).

The difference ΔC between these costs is

$$\Delta C = C_1(I) - C_2(I) = \sum_{k\in I} \sum_{j=1}^{n} \Psi_j d'_{ji} - \sum_{j=1}^{n} \Psi_j d'_{j1}$$

$$- \sum_{k\in I-\{1\}} \sum_{j=1}^{n} \Psi_j d'_{1k} = \sum_{k\in I-\{1\}} \sum_{j=1}^{n} \Psi_j (d'_{jk} - d'_{1k})$$

As it turns out, and will be shown later in an example, ΔC could be either positive or negative. That is, a non-primary model is not necessarily cheaper than a primary one, and vice versa. A given site j (which is a source

A given site j (which is a source of updates) will contribute positively to ΔC if $\sum_{k\in I-\{1\}} d'_{jk} > \sum_{k\in I-\{1\}} d'_{1k}$ and negatively if $\sum_{k\in I-\{1\}} d'_{jk} < \sum_{k\in I-\{1\}} d'_{1k}$. Both cases are possible.

In table VI-1 we present C_1 and C_2 for the five-node example that Casey introduced in [C1]. (See section II.2 for the data and some discussion about Casey's example.) In the table we have five alternatives for C_2 according to which site is chosen to be the primary. The optimum for three of the five primary alternatives is more expensive than the optimum for C_1.

In appendix D we actually prove that the theorems that Casey presents [C1] are valid for the primary model presented here. The only nuisance is that the primary location must be preselected. Future work could be directed toward finding ways to reduce the number of primary candidates. However, given that the problem grows exponentially (like 2^n), the extra factor of n needed to try each site in turn as the primary is not precisely the first factor that we should try to reduce.

Non-primary Model		Primary Model (Bunch)				
I	(Johnson)	p=1	p=2	p=3	p=4	p=5
1	960	960				
2	972		972			
3	1030			1030		
4	918				918	
5	915					915
12	852	810	822			
13	774	876		882		
14	726	807			765	
15	867	882				837
23	856		822	816		
24	730		888		834	
25	735		819			762
34	804			816	768	
35	729			882		831
45	753				768	765
123	810	870	744*	876		
124	762	801	882		897	
125	759	732*	813			756
134	756	939		876	759	
135	753	870		1014		825
145	705*	801			759	687*
234	760		882	738*	828	
235	765		813	876		894
245	717		951		828	756
345	711			876	680*	825
1234	792	933	876	870	891	
1235	789	864	807	1008		888
1245	741	795	945		891	750
1345	735	933		1008	753	819
2345	747		945	870	822	888
12345	771	927	939	933	885	882

* optimum for this model

Table VI-1

Comparison between a primary and non-primary model for the 5-node example in [C1]. Entries are costs: $C_1(I)$ for the non-primary model and $C_2(I)$ for the primary model. The column headings "p=i" indicate that the primary is at site i.

Chapter VII. CONCLUSIONS

First of all, we have shown that distributed data base systems are useful and desirable for many environments. A distributed data base system can improve availability and response time. In addition, even when network transmission costs are included, a distributed system can be cost-effective (with respect to operational cost). Many researchers, having realized these facts, have made respectable contributions to the relevant literature [A1, A4, B4, C1, C2, C3, C4, D2, L2, etc.].

A great majority of the researchers in related areas assume that all data base updates are sent directly from the site generating the update to all sites which have a local copy of the data base (a non-primary model). We have shown that such an approach is feasible only in very restricted situations. In particular, an important condition seems to be that no update modify the fields used for the selection of records. For example, "if salary = x, then change salary to y" is not an acceptable update.

Our principal goal was to present the characteristics of a workable model. Clearly a non-primary model meets this goal in a very restricted set of situations. However, in those environments in which the power (as defined in IV.6) of this model is sufficient and the inherent restrictions (e.g., memory) are acceptable we could be better off (i.e., obtain better throughput) by choosing a non-primary model. A Johnson-type non-primary model has the added advantage that it is expected to be easy to implement.

A major consideration for the application manager intending to use a Johnson-type model will probably be the synchronization of the clocks used by the different data base managers. Clock synchrony seems to be, in general, a desired feature. However, tolerance to clock asynchrony is very dependent on

the particular application. It is easy to imagine an environment in which some sites should have higher priority than others. This could be achieved by setting the clocks purposely wrong, giving a higher value to data base managers with higher priority.[1]

If clock synchrony is desired, we could make use of one of the following options: a hardware solution, the Lamport scheme (section V.1), or our synchronization protocol of section V.1. Alternatively, we could avoid the need for synchronized clocks by using the Reservation Center model.

A hardware solution could be obtained by using redundant clocks. (Care must be taken to protect against wrong clock settings by operators.) Any of the other three alternatives are more sensitive to failures, such as partitioning. Each of them has some advantages and disadvantages which could help an application manager make his choice. Lamport's scheme is easy to implement but is susceptible to clock drifting in the positive direction. Our synchronization protocol of section V.1 is more complex to implement, has some unappealing ways of correcting situations in which multiple clocks are incorrectly set back, but seems to be capable of maintaining the clocks in reasonable synchronization with respect to the real time. The Reservation Center model RC(m,n) also has a relatively complex protocol, but obtains a synchronization on the order[2] of mn by eliminating clocks and replacing them by a centralized synchronizer.

In short, we have presented several versions of Johnson's model in the hope that a vast majority of the potential users of that model will find one of the given alternatives suited to their needs.

---

[1] Note that maintaining this pre-established clock asynchrony then becomes an important factor. It can be done by a scheme almost like that for maintaining perfect synchronization.

[2] By order of mn we mean that an update entered mn time instances after another one is with a very high probability going to get a bigger ticket number.

For those applications which require more power, we have presented a primary model (introduced by Bunch [B5]). Under normal circumstances, all the data base managers in a primary model will be able to order the updates correctly before application and hence to automatically apply them in the correct order. This uniform application order allows any kind of operation to be supported. Thus, primary models are as powerful as we can get.

The transition from a non-primary to a primary model for environments that could support either is not cost free. Overall throughput and other factors are slightly degraded (sections IV.5 and IV.7) to obtain the additional power. In addition to more power, some other advantages, such as better memory utilization, are gained in the shift (section IV.7).

After convincing ourselves that, in general, a primary approach is the right one, we undertook the job of making it workable. Bunch's model [B5] is very sensitive to multiple failures (e.g., the missing update problem). Making a system resilient grows harder as the number of failures that we want to be able to successfully overcome increases. However, as failures grow more complex, involving more components, the probability of their occurrence is drastically reduced. In order to make a distributed data base system workable we have presented a compromise. Rather than the impossible goal of complete resiliency, we presented an n-host resilient model (section V.3), n being the number of simultaneous failures which would be needed to disrupt the service. A disruption of the service will not affect the consistency of the system; it will only degrade the service (possibly to a query-only state). Given that the parameter n in the n-host resilient model is arbitrary, the application manager could decide on its value in accordance with his own needs.

In summary, we have shown that distributed data base systems are useful. We have presented and evaluated the available models capable of

supporting such a system. We have studied the appropriate environments suitable to each one of these models. Finally, we have presented workable versions of each one of the models. Thus, our original goal has been satisfied.

With regard to future work, the obvious extension is the actual implementation of a resilient primary model (for example, our broadcasting model). The complete and extensive specification of all the required details of such a model seem a likely step for the near future. It is not expected that this implementation will be a simple matter. Many small problems will have to be solved. We have tried to present all the substantial problems that might arise in such an implementation. However, our limited human nature is capable of overlooking some of them. Some other problems will probably not appear until the actual implementation starts to be operational.

A similar statement could be made for the implementation of the clock synchronization schemes of section V.1.

Once we succeed with the implementation of a distributed data base system, it would be of great interest to study the interaction between files and programs in such a system. Once the file is in multiple locations it seems very plausible that transferring complete jobs to another computer system will be sometimes economical. Savings could be due to a better utilization of the specific characteristics of each computer in a heterogeneous network [A1, A2] or due to load sharing [B1]. Network multiprocessing is a possible outcome of such a study. It is possible that it could prove to be worthwhile to transmit not only complete jobs but incomplete ones as well. We could actually be watching the beginning of a network multiprocessing operating system.

Many interesting problems remain to be studied in the general area of distributed data base systems. For example, strategies for improving or optimizing response need to be developed. In general, the very difficult topic of

response time in a distributed data base system has not been extensively addressed. Chu [C5] is the only one to attempt a response-time analysis, but he makes a lot of simplifying assumptions of questionable validity.

In the area of file allocation there are many open problems. As we point out in appendix A, Levin's solution of the program-file allocation does not yield a true optimal allocation for the programs. Obtaining some insight into this problem could easily be of some practical value. Many extensions are foreseen in the near future for the various ideas that we have introduced in the file allocation discussion of appendix A. Our suboptimal search algorithm has given some optimistic results, but it needs more thorough testing. The same is true for our a priori conditions. They have performed very well for the examples presented in the available literature, but a broader spectrum of results is desired.

Appendix A. FILE ALLOCATION

Given the high degree of interaction between the file allocation problem and distributed data base systems, we find it advisable to include this appendix in order to properly present the state of the art of file allocation. In so doing we will also include some contributions that we have personally made.

A.1 The Problem

Once our availability and response studies indicate that we should take advantage of duplicate files, the immediate question is: Where? Where should the copies of a file be allocated in order to minimize some given cost function? The following lines are intended to assist us in answering this question.

In general a file allocation model has two types of transactions: updates and queries. The traffic required for such transactions through a given network depends (among other things) on the number of copies of the files. Since a given update must be seen by (i.e. sent to) all the copies of a file, the more copies we have the higher the update traffic will be. This traffic will be minimized if there is only one copy of each file. On the other hand, the addition of new copies of a file tends to reduce the network query traffic, up to the point where every site has its own copy of the data base and responds to its queries locally, so that there is no network query traffic. Clearly there is a tradeoff. We need to introduce a cost criterion and optimize a given system with respect to that cost.

Before we get into the details of how to objectively establish an optimal allocation, we should point out that the optimum is not always the best from all points of view. In real life we should be ready for some human

antiefficient subjectivity, such as: I want a copy HERE! I don't want him to have a copy, he is not my friend! Either John or Tony might have a copy but not both! These and other factors force upon us the consideration of restricted environments (section A.7), if the constraints imposed allow more than one feasible alternative.

A.2 Casey's model and theorems

The most common cost criterion used in the literature is an operational cost, in which update and query traffic as well as storage costs are considered. Casey [C1] gives us a typical example of such a cost function:

$$C = \sum_{j=1}^{n} \left[\sum_{k \in I} \Psi_j d'_{jk} + \lambda_j \min_{k \in I} d_{jk} \right] + \sum_{k \in I} \sigma_k$$

Where:

I = index set of sites with a copy of the file

n = number of sites in the network

Ψ_j = update load originating at site j

λ_j = query load originating at site j

d_{jk} = cost of communication of one query unit from site j to site k

d'_{jk} = cost of communication of one update unit from site j to site k

σ_k = storage cost of file at site k

$\lambda_j \min_{k \in I} d_{jk}$ = cost of sending query to "closest" copy - or one that it is cheapest to transmit to

$\Psi_j d'_{jk}$ = cost of sending update to the k^{th} copy

Once this cost function has been established, the optimal allocation (i.e. the set I which minimizes C(I)) could be obtained by several different approaches. As Casey mentions, this problem is analogous to the factory location problem found in transportation (operations research) literature.

The classical solution of this kind of problem makes use of mixed integer-linear programming. Chu [C5] has presented a zero-one linear programming formulation of this problem.[1] In general, this kind of technique turns out to be computationally very expensive. On many occasions heuristics are used, sacrificing optimality for computational cost. For example, Levin [L2] points out that Chu's approach would produce about 9000 zero-one variables with 18000 constraints for an ARPA-like network with 30 sites and at least 10 files. There are inherent difficulties for large systems, but it should be pointed out that this approach is mathematically straightforward. Many restrictions can be added just by appending the right constraints.

Casey's alternative to the integer-programming approach is a heuristic search based upon the following two theorems [C1].

Theorem 1. Let $d_{jk}=d'_{jk}$ for all j,k. If for some integer r ($\leq$n), $\Psi_j>\lambda_j/(r-1)$ for all j, then any r-site file assignment is more costly than the optimal one-site assignment.

Theorem 2. Suppose assignment I is optimal. Then along any path in the cost graph from the null node to the node corresponding to I, cost is monotonically non-increasing from one node to the next.

Actually Casey's Theorem 2 is somewhat more general and is not stated in terms of the notions of "paths" and "nodes". These notions belong to the "cost graph" visualization of the optimal allocation problem. (See figure A.2-a.) At level i in the graph we have all the allocations of i copies (each node representing an allocation) with a branch from a node at level i to one at

---

[1] Actually Chu [C5] solves a slightly different problem, but one which is very easily modifiable to the current one. Chu assumes that all transactions (updates or queries) return some information to the user and that, if we are dealing with an update, a modified version of the original data is sent back to the data base.

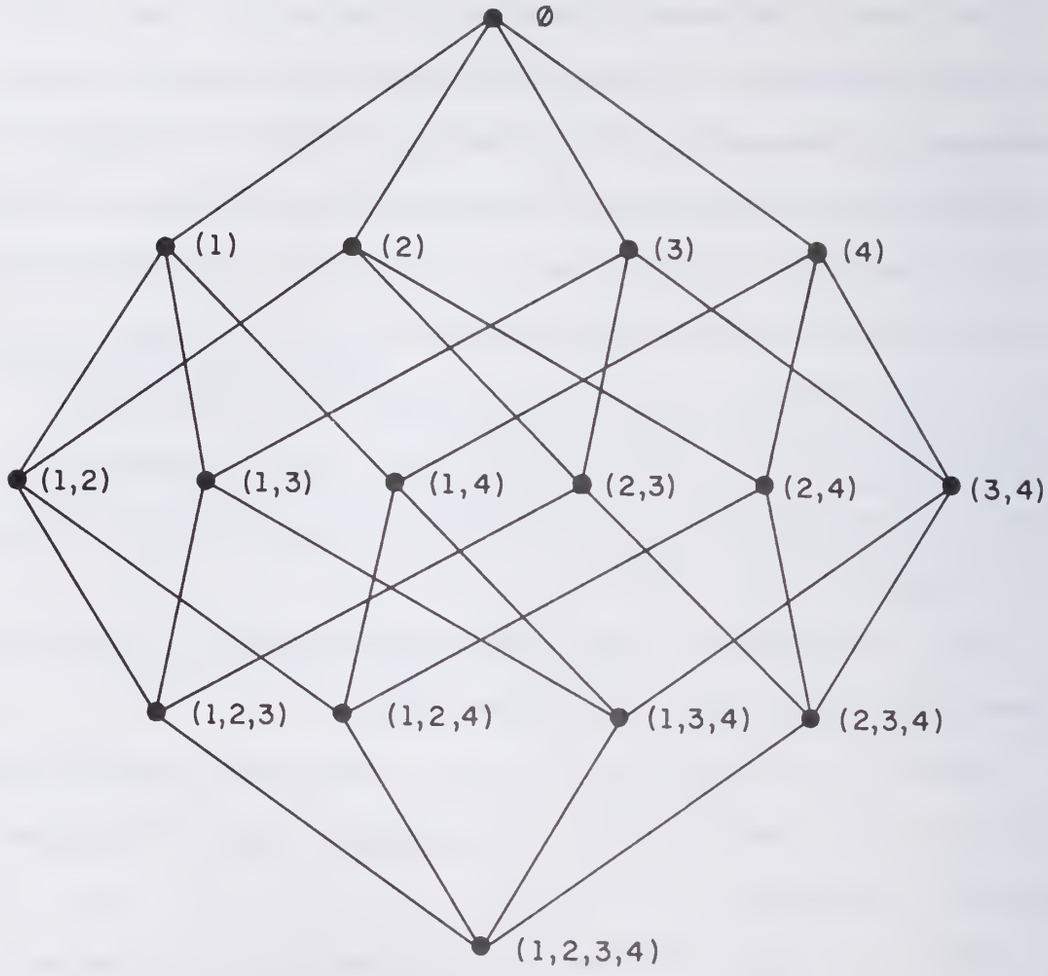


Figure A.2-a

Graph of all possible allocations
among four sites.

level i+1 if and only if the latter allocation can be obtained by adding a single copy to the former.

Theorem 1 effectively gives us an upper bound (r) beyond which there is no need to search; i.e. a gross stopping criterion. Theorem 2 gives us a more precise stopping criterion. If while following a path we find that the cost increases then no further search along this path will be of any use.

In Casey's words, "A computer algorithm can be implemented in several different ways to select file nodes [sites] one at a time up to the optimum configuration. One approach is to follow all paths in parallel through the cost graph, stepping one level per iteration. This method is computationally efficient, but may require a great deal of storage in a large problem. Alternatively, a program can be written to trace systematically one path at a time. Such a technique uses less storage, but may require redundant calculations since many different paths intersect each vertex". In general we agree with this statement, but we dispute the last sentence. We believe that no node must be visited more than once in a "path at a time" approach. In a later section (A.6), we will defend our belief.

It should be noticed that Casey's solution is actually an intelligent and ordered enumeration that will in some cases result in a complete enumeration. (For example, if the minimum is obtained by putting a copy everywhere, the application of theorems 1 and 2 will not reduce our search).

A.3 Other related work

Levin's Ph.D. Thesis [L2] is presented as an extension of Casey's work. In Levin's work, files and programs are separated and a staged minimization approach is tried. The staging may be described by:

$$\begin{array}{c} (1) \\ \text{Min} \\ \text{file location} \end{array} \left\{ \begin{array}{c} (2) \\ \text{Min} \\ \text{program location} \\ \text{(given file location)} \end{array} \left\{ \begin{array}{c} (3) \\ \text{Min} \quad \text{(cost function)} \\ \text{routing} \\ \text{(given file and} \\ \text{program location)} \end{array} \right\} \right\}$$

When we carry out the minimization (3), we get the expected answer: Send all queries to the closest copy, and send all updates everywhere they are needed. To solve minimization problem (2), Levin assumes that storage costs for programs are zero. This implies that we store programs everywhere we are allowed to (and can find any slight advantage to doing so). Finally, minimization (1) follows Casey's ideas.

It is our belief that the idea of separating files and programs could be useful in the light of Alsberg's [A1] discussion on disparity of processing costs in the ARPA network.

Levin goes on to use similar ideas to describe other environments where the transmission rates are not known or when they change dynamically with time. These seem to be valuable considerations.

Going into the details of Levin's procedure, we find the minimization (2) is a little tricky. It actually solves the problem by avoiding it. The assumption that program storage has zero cost (regardless of the rationale behind such a decision) implies that programs are basically stored everywhere. In this circumstance, program locations are chosen a priori by the designer. This contradicts Levin's argument that it is necessary to minimize the number of copies of a program due to the problems of maintaining updated versions of a program in an heterogeneous network. Furthermore, this type of program allocation could be obtained immediately from Casey's model if some additional precomputing is done to reflect a change in the routing technique. Each

transmission from one node to another will have to touch at least one of the preestablished program sites. Thus, if the original $\hat{d}_{ij}$ does not touch any such sites, then it should be modified to $\hat{d}_{ij} = d_{ik}+d_{kj}$ where k is the program site which minimizes d_{ij}.

Urano et. al. [U1] establish a concept of proximity. Suppose that two sites A and B have the following property: Given that we have a copy at one of them (let's say B) the cost of referring to B every transaction that arrives at A is smaller than the cost of having a second copy at A. In this case,[1] no optimal solution will include both copies. To state this result formally, first define ρ_x as

$$\rho_x = \frac{\sum_{i=1}^{n} \psi_i d'_{ix} + \sigma_x}{\sum_{i=1}^{n} \lambda_i} - \delta_x = \frac{Z_x}{Q} - \delta_x$$

where δ_x is the relay cost at x (cost experienced by site x if the only thing it does to a message is to send it somewhere else), Z_x is the storage and update cost of having a copy at x and Q is the total query load. Then: [Urano's Property 2].

In an optimal allocation there is no pair of sites (A,B) such that $\rho_a > d_{ab}$. That is, if $\rho_{AB} \equiv \text{Min}(\rho_A, \rho_B) > \text{Max}(d_{AB}, d_{BA})$ then A and B will never simultaneously be in an optimal allocation.

The above result follows from the fact that if $Q(C_{xy} + \delta_x)$ is less than Z_x then it is cheaper to send everything to y rather than have a second copy at x.

---

[1] Some additional assumptions are made in [U1]. We omit them because no clarity is gained by including them.

This approach seems useful when we have a large number of sites. The exhaustive search of an n-site network would require the evaluation of up to 2^n costs. But suppose the sites are partitioned into proximity classes with P_i elements each ($1 \leq P_i \leq n$) and such that only one member of a class can be included an optimal allocation. Then the number of costs that need to be evaluated is $\pi(P_i+1)$, where P_i+1 is the number of ways to pick one element from the class, plus one (for the case of no element). For example, if we have a network in which proximity is found for a 3-site conglomerate ($P_o=3$, $P_1=P_2=\ldots P_{n-3}=1$), then instead of 2^n evaluations we need $2^{n-1} = (P_o+1)(P_1+1) \ldots = 4(2)^{n-3}$.

For completeness, we will also mention Chang [C3] who deals with file allocation in a hierarchical (treelike) network. His basic contributions are some criteria to allocate files when designing a network with nonlinear transmission and storage costs. (The non-linearity is with respect to amount of traffic and amount of storage respectively.)

A.4 How critical is the evaluation problem?

Enumerating and evaluating all alternatives for a network with, let's say, 19 sites seems out of reach. However Casey points out that a program run on the 360/91 took less than 10 seconds to solve six different single-file optimal allocation problems for a network of 19 sites. (This time included the Fortran compilation.) This result tends to show that the optimization of allocation in networks of current sizes is quite manageable following Casey's guidelines. Further work might seem to be only of theoretical interest. This is unhappily not true.

Let's assume that in Casey's 360/91 it takes 2 seconds to solve a 19-site problem. Furthermore, since each additional site basically duplicates the number of alternatives for the allocation we will assume that the computational time also doubles each time the size of the network increases by one. This

implies that to compute a 30-site allocation would require around 68 minutes, a 40-site allocation would require over 48 days, and a 50-site allocation would require about 136 years. The times involved increase exponentially with the number of sites. (See Eswaran's [E1] proof that this problem is polynomially complete.) Even for moderate-sized networks heuristic aids are badly needed.

A.5 Our contribution

In the light of the discussion in the previous section, it seems reasonable to try to reduce as much as possible the number of cost-graph nodes to be evaluated. We would clearly use Casey's theorem 1 and 2 for such purposes. Urano's condition 2 would probably be used as well. We now present three additional conditions that might sometimes be handy in reducing our search. We assume throughout (as Casey does) that $d_{ii} = 0$ for all i. The cost of storing and updating a copy at site i is an important quantity; we shall denote it by Z_i. That is, we let $Z_i = \sigma_i + \sum_{j=1}^{n} \psi_j d'_{ji}$.

Condition 1. Unquestionable inclusion. A site A should unquestionably be included in an optimal solution if

$$\lambda_A \min_{B \neq A} d_{AB} > Z_A \qquad \text{A.5-i}$$

That is, site A should unquestionably be included in any optimal allocation if the cost Z_A of having a local copy is smaller than the minimum cost of having to send the queries elsewhere. A formal proof of this condition is given in appendix B.

This very simple condition would have actually reduced Casey's 5-site example [C1] to a 3-site one. In Casey's example, site 4 and site 5 should be unquestionably included. For site 4 the left side of A.5-i is 144 as compared to 126 for the right side. Similarly, for site 5 the left side is 144 as compared to 123 for the right side.

Given that we know that sites 4 and 5 must be included, it is only a question of deciding whether to include sites 1, 2 and 3. Thus, a program similar to the one presented in appendix C will have to evaluate costs for only 7 nodes (2^3-1) as compared to 31 (2^5-1) if this condition is not used.

Condition 2. Unquestionable exclusion. A site A will not be included in any optimal solution containing more than one site if:

$$Z_A > \sum_B \lambda_B \left(\max_C d_{BC} - d_{BA}\right) \qquad \text{A.5-ii}$$

That is, a site A should not be included in an optimal allocation if the cost of having a copy at A is bigger than the greatest possible savings that we could obtain. A formal proof is provided in appendix B.

Conditions 1 and 2 can be summarized as follows. Let

$$m_i = \lambda_i \min_{j \neq i} d_{ij}, \text{ and}$$

$$M_i = \sum_{j=1}^{n} \lambda_j \left(\max_k d_{jk} - d_{ji}\right).$$

Then for each i the real line is partitioned by m_i and M_i into three regions, as shown in figure A.5-a. If Z_i falls in region 1, then it should unquestionably be included in any optimal allocation. If it falls in region 3, it will generally be excluded. An exceptional case is when all sites fall in region 3; i.e., satisfy inequality A.5-ii. In this case, all optimal allocations will be single-copy ones, and we need only choose the best of these. Sites whose costs Z_i fall in region 2 (including the boundary points) must be considered further.

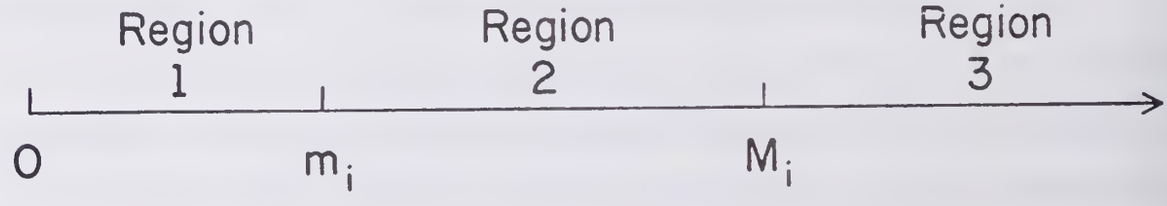


Figure A.5-b

Partitioning of real line into three sections using m_i and M_i as delimitors

One other comment should be made. If network transmission costs are independent of the sites involved, so that $d_{ij}=d$ for all $i\neq j$, then $m_i=M_i=\lambda_i d$, and region 2 collapses to a single boundary point. Unless the cost Z_i of some site lies on the boundary, every site is a priori either included or excluded. Immediately we see that then the set of sites in region 1 forms the optimal allocation, unless the exceptional case noted above obtains. Computation is therefore reduced to a minimum (essentially n cost evaluations) for many real network environments. (For example, most commercial packet-switched networks charge per packet, irrespective of the distance the packet is to be sent.) There are, of course, real cost differentials incurred in the longer lines between sites, but from the user's point of view it is the network charging policy that matters.

Condition 3. Neighborly excluded. If we have two or more sites that are relatively close, this condition could guarantee that one will always be excluded from an optimum. Suppose that for two sites (A and B) all possible savings that could be obtained by using B instead of A are offset by B's being more expensive than A. That is

$$\sum_{j'} \lambda_{j'}(d_{j'A} - d_{j'B}) < Z_B - Z_A \text{ where } j' = \{j \mid d_{jB} < d_{jA};\ j=1,\ldots,n\}$$

Then, if in a given allocation only one of them (B) has a copy, changing the allocation from B to A will always produce some savings. However, it is also possible that having both copies is cheaper than having only one. That is, after allocating a copy to A it is still possible that allocating a second copy to B could introduce some additional savings. This does not happen if $\sum_{j'} \lambda_{j'}(d_{j'A}-d_{j'B}) < Z_B$ (j' as above). Thus we obtain (see appendix B for a complete proof).

Condition 3: A site B is neighborly excluded from a given optimization problem if there exists a site A in the network such that:

$$\sum_{j'} \lambda_{j'}(d_{j'A} - d_{j'B}) < Z_B - Z_A$$

where $j' = \{j \mid d_{jB} < d_{jA},\ j = 1, \ldots, n\}$

This condition seems very similar to Urano's proposition 2. However, we feel that it is in some sense stronger. It tells us when a particular site cannot be in an optimal solution and not that only one of a given set could be included in a given solution.

An actual implementation will probably not try every possible pair of sites to establish neighborly exclusion. Some kind of criterion should be used to locate possible candidates. (For example, for each site A_1 one might identify the site A_2 for which a few of the biggest $d_{A_1 j}$ are close to $d_{A_2 j}$).

If we apply condition 3 to the ARPA example in Casey's [C1] paper we find that 6 of the 19 sites should be neighborly excluded (sites 3, 4, 5 by 2; site 15 by 14; sites 17, 19 by 18). This reduces the maximum number of cost nodes to be evaluated by a factor of 64.

To conclude this section, we would like to point out that the evaluation of Z_A required for testing the three conditions is never a waste of time. In the program shown in appendix C this value is assumed to be precomputed in order to speed up the evaluation of a node cost. Thus, checking if condition 1 is satisfied might well be worth the effort, especially since any success reduces the maximum number of node evaluations by 50%.

The additional computations required for conditions 2 and 3 might be too expensive. Some selective checking (for example, when Z_A is big try condition 2, etc.) might be advisable.

A.6 A program to search for an optimum in Casey's model

Casey [C1] gives two suggestions for how to go about searching for the optimum assignment: 1) level by level, and 2) one path at a time. A level-by-

level approach, as he pointed out, is computationally efficient but very expensive in storage cost. On the other hand, the one-path-at-a-time approach requires little storage but is computationally inefficient. As Casey points out, given that a node is visited by various paths, it might be required to evaluate its cost more than once. It is our intention to present a third alternative which will make moderate use of memory, while the computational efficiency is maintained.

If we neglect Casey's theorem 2 (see section A.2), the level-by-level approach would have to perform an exhaustive search. However this approach uses little memory. If we make use of the theorem, we would obtain a computationally more efficient algorithm, but at the same time it would be very expensive in memory. We need at the very least $\binom{n}{n/2}$ memory locations (for n even) to save the values of the previous level (this maximum occurs in the "middle" of the cost graph) and some additional working space for the evaluation of the current level. Note that $\binom{n}{n/2}$ for n=20 is 184,756 and is 155,117,520 for n=30. We will use only O(n) locations with the same computational efficiency. Actually the program presented in appendix C uses $O(n^2)$ locations for efficiency purposes.

The essence of our method is to search one path at a time, but by a scheme in which each node has only one successor, and thus is visited only once.

The search strategy for five sites would start: 1, 12, 123, 1234, 12345, 1235, 124, 1245, 125, 13, ..., 5. The next algorithm will print out this order:

0. Let V(i) be an array with index i. Let the instruction "print" be such that it will print the values of V(i) from V(1) to V(ℓ) contiguously.

1. We start with site 1. V(1)=1; ℓ = 1; print;

2. If V(ℓ)$\neq$n (then we can increase the level by 1), then

ℓ= ℓ+1; V(ℓ)=V(ℓ-1) +1; print; go to 2;

3. (If V(ℓ)=n then we can't extend this path, so we go back and increase the previous element in the path).

ℓ = ℓ-1;

If ℓ=0 then halt. (We are done.)

V(ℓ)=V(ℓ)+1;

print;

go to 2;

What we have actually done is to convert the cost graph shown in figure A.2-a to the tree shown in figure A.6-a. The above algorithm follows a preorder search of this new tree.

As the reader will observe, there are many paths that take us to the node 12345 in figure A.2-a, but only one in figure A.6-a. The question now is: Is it possible that by following only one path to a given node that we will miss the optimum? That is, could node 12345, for example, be an optimum and yet not behave like an optimum when the search is along the unique path shown in figure A.6-a? The answer is no!

In order for any node to be the optimum it must, by definition of optimum, have a smaller cost than any other node, in particular than any node in the prior level. This fact in conjunction with Casey's theorem 2 produces the condition that along "all" paths leading to the optimum node, cost is non-increasing. Thus, our simplified tree of figure A.6-a contains one such path which will lead us to the optimum. We thus conclude that our restricted search is sufficient to detect the optimum.

As the reader will realize, our algorithm requires in the worst case one memory location per level, i.e. n memory locations.

In the program of appendix C, for each level along the search path we keep a vector of the associated query costs. This allows us to quickly establish

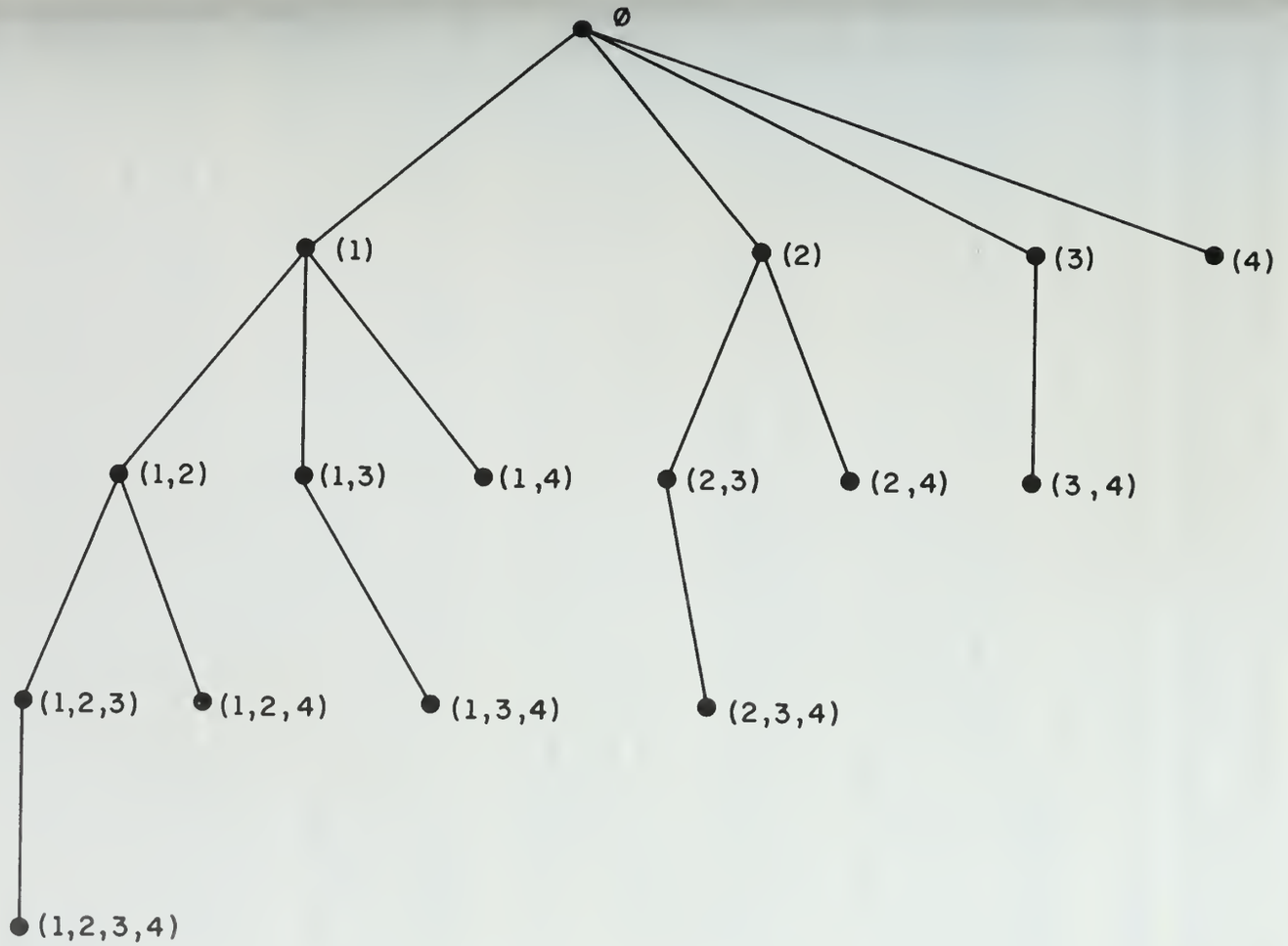


Figure A.6-a

Rearrangement of four-site allocations (figure A.2-a) into a tree for efficient preorder search.

the query transmission savings that a new allocation (adding or changing a site) will produce (by comparing the values of this vector with the cost that must be paid if the queries are sent to the new copy). These vectors increase our memory requirement to (n^2+n), or simply to $O(n^2)$, but the run-time could be substantially reduced.

A.7 Restricted environments

In many real life situations external conditions impose restrictions on our models. For example: to improve availability we could be required to have more than a certain number of copies regardless of cost. Security considerations could forbid one site from holding a copy while requiring another to have one. The average response time for queries might be required to be lower than some upper bound.

For our discussion we will divide the possible constraints into 3 categories: (i) level constraints, (ii) site constraints, and (iii) other constraints.

(i) Level Constraints. If the levels (specifying number of copies) for which the optimization is to be carried out is restricted, (for example if availability considerations impose a lower bound on number of copies), then the techniques described before are not always useful.

If we could establish that the overall (global) optimum has not been eliminated by such a restriction, then all the results given in previous sections are valid here. Actually, some minor modifications could make the program in appendix C work more efficiently in this case by starting the search at a higher level.

When we know that the restricted optimum is not a global optimum, then Casey's theorem 2 is useless. However, conditions 1, 2 and 3 could be still used if they do not drive us out of the feasible space (i.e., do not

unquestionably exclude more than the number of nodes that should be excluded). But even if they drive us out of the feasible space, this fact could give us a great deal of information. (That is, if the number of sites "unquestionably excluded" is too big, we know that all the others should be included and we should choose the remaining ones from the unquestionably excluded ones).

In general we do not know if the global optimum is included or excluded in a restricted problem. In these circumstances, an exhaustive search of all feasible solutions might be needed, especially if the feasible space is relatively small. Alternatively, we could try one of the heuristic techniques in (iii), below.

(ii) Site Constraints. If we are given this type of constraint, we should in general be thankful. These restrictions tend to reduce the size of our problem. Actually, conditions 1, 2 and 3 and Urano's property 2 impose this kind of restriction; and by so doing reduce the problem.

(iii) Other Constraints. Casey's model is not suited to all kinds of restrictions. If response time is a concern, it is not immediately clear how to incorporate it. Furthermore, if we follow Chang's idea [C3] of having a non-linear cost function or Levin's suggestion [L2] of a dynamic query and update load, Casey's model starts to look inadequate.

Rather than destroying Casey's model we will describe an heuristic algorithm that could be used in a few of these cases to locate sub-optimal solutions. Other cases will be briefly discussed in our conclusion section. We take this approach because we believe that Casey's model is adequate for most purposes.

In many circumstances, Casey's approach could be unreasonably expensive even with the assistance of the theorems and conditions given above. In many other cases, due to imposed restrictions, it might be impractical to try

Casey's approach. In all these cases, we could use the following algorithm that will lead us (in general) to a sub-optimal local minimum.

The algorithm is based on the idea of letting the cost "roll down" along a path in the cost graph, up to the point where no more rolling down is possible. At that moment we are at a local minimum.

To implement the algorithm we start with any feasible node and start testing the neighboring feasible nodes until we find one with a lower cost. At that point we roll down to that node and start all over again, up to the point where no more rolling down is possible; i.e., all feasible neighbors are more costly. Then we are at a local minimum.

It is possible that the cost of a node will be evaluated more than once. However, a roll down to a node that has been evaluated more than once can not occur. Thus there is no danger of an eternal loop. For example, consider the 3-site network of figure A.7-a.

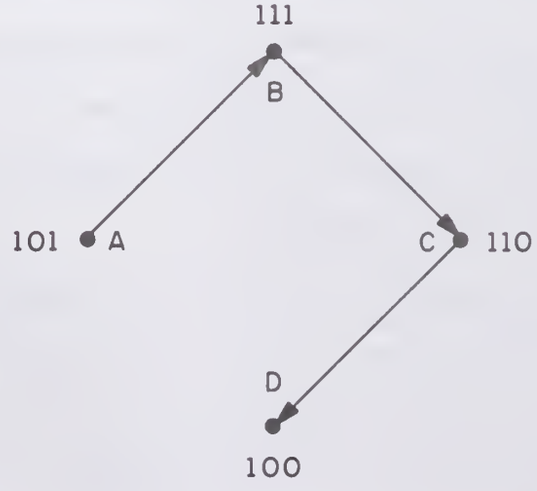


Figure A.7-a

A piece of a three-site cost graph. Costs C are such that C(A) > C(B) > C(C) > C(D). D is a local minimum. Allocation A corresponds to a file at sites 1 and 3 (101), B to a file at the 3 sites (111), etc.

If in figure A.7-a we start with node A (copies at sites one and three) and follow the search through B, C and D, we would probably require that site A be reevaluated when we are at site D in order to establish the local optimality of D.

The reader could then argue that this method could be very costly, since nodes might be evaluated more than once, and he is right. To avoid this we could have a flag for each node, and turn it on when an evaluation is made. This approach could be costly in memory. (2^n flag bits might be required.) Alternatively we might try to reduce the probability of duplicate node evaluation by choosing the nodes in such a fashion that we always try to go away from recently visited nodes. The following algorithm implements such a search.

A.8 Suboptimal search algorithm

The search will be performed by changing the allocation by one site at a time. (If the site has a copy, we eliminate it; otherwise we allocate a copy there). In order to reduce the probability of multiple evaluations, we will try to make this change in every other site before again trying to change the situation of a given site. Thus, when we arrive at a new node, we will try to get as far as possible (up to n nodes away) by modifying all specific allocations that lead us to that node. If we do not succeed in getting very far (so that the probability of a reevaluation is bigger), it is probable that we are close to the local minimum, and the number of such reevaluations would hopefully be small.

In order to do the search in an organized way we will make use of a circular list (CLIST) with all the site numbers contained once and only once in it. We will then go around the circular list trying to alter the allocation state (i.e. whether or not the site has a copy) of each site until we reach a local minimum.

A local minimum is located if we are able to make a complete cycle of the circular list with no cost improvement. That is, if we have a node which represents a local minimum, then all adjacent nodes will have a higher cost. To keep track of this, each element of the circular list will have a counter (CLISTCOUNTER) that will indicate the value of ROLLCOUNT (the count of rolldowns) the last time that the alteration of the allocation of this site was tried. When, while cycling the list, we get to a site whose CLISTCOUNTER is the same as the current total of rolldowns (ROLLCOUNT) we know that we are done; i.e., we are in a local minimum.

The performance of the algorithm will vary somewhat with the order of the sites in the circular list. There are n! alternatives to choose from. Clearly we do not know which one is the best search strategy, and some guess or random choice will be made.

Following the ideas of conditions 1 and 2, we could establish a search strategy with a somewhat higher probability of leading us to the optimum. We could say that the more queries (λ_A) a given site (A) generates the more likely it is that we will save if we allocate a copy of the file to that site. Similarly, the lower the cost of having a local copy (Z_A in section A.5), the more likely it is that it will pay to have a local copy. We then define a search factor $S_{A_i} = Z_{A_i}/\lambda_{A_i}$ to be used to establish the search strategy. In our circular list, the sites will be listed in order of increasing S_{A_i} values (i.e., A_i goes before A_j if $S_{A_i} < S_{A_j}$). The site with smallest S_{A_i} will be pointed at by the top-of-list pointer (LISTPOINTER) at the beginning of the algorithm.

We point out that this search strategy is consistent with our previous results. The smaller S_{A_i} is, the more likely it is that A should be unquestionably included. The larger the S_{A_i}, the more probable it is that we have a

site that should be unquestionably excluded. Similar search factors ($S'_{A_i} = Z_{A_i}/\lambda_{A_i} \min_{B \neq A_i} d_{A_iB}$ and $S''_{A_i} = Z_{A_i}/\lambda_{A_i} \max_{B} d_{A_iB}$) could be used as well to reflect a little closer our results in section A.5. However S'_{A_i} tends to be big for "all" nodes that are close together and S''_{A_i} tends to be small for isolated nodes, facts that are not very welcome. (Some alternatives, such as $S_{A_i}/\bar{d}_{A_iB}$ where $\bar{d}_{A_iB}$ is the average distance from A_i to all other sites could be tried). Our algorithm (for an unrestricted environment) will look like the following.

0. Store in CLIST the search strategy (i.e. the site ordering) and make LISTPOINTER point to the first element. Initialize all CLISTCOUNTERS to -1, MINIMUM to infinity, ROLLCOUNT to zero. Let ASSIGNMENT = null.

1. IF ROLLCOUNT = CLISTCOUNTER(LISTPOINTER) THEN HALT
   We have completed a full cycle with no roll downs; i.e., we are at a local minimum and done.

2. NEWASSIGNMENT = ASSIGNMENT + Change Status of Site (CLIST (LISTPOINTER)) (i.e. if the site has a copy eliminate it, otherwise allocate a copy there)

3. IF COST(NEWASSIGNMENT) < MINIMUM THEN (we will roll down)
   ROLLCOUNT = ROLLCOUNT +1;
   MINIMUM = COST(NEWASSIGNMENT)
   ASSIGNMENT = NEWASSIGNMENT

4. CLISTCOUNTER (LISTPOINTER) = ROLLCOUNT;
   LISTPOINTER = (LISTPOINTER+1) mod n (Advance pointer to next element in circular list).
   Go to 1.

Using Casey's [C1] 5-site example to test this algorithm, we obtain the following results. If we use the search Factors S_A to order our search strategy, the order is 5 ($S_5 = \frac{123}{24} = 5.125$), 4 ($S_4 = \frac{126}{24} = 5.25$), 1($S_1 = \frac{168}{24} = 7$), 3 ($S_3 = \frac{174}{24} = 7.25$) and 2 ($S_2 = \frac{180}{24} = 7.5$). With this ordering we hit the optimum right on the nose after 3 roll downs. That is, after 7 iterations of our algorithm (3 to get there and n-1=4 to discover that it's a local minimum) we are done.

If we vary the search strategy we find that in 12 cases (of the 5!=120) we get to a local minimum in 7 iterations, in 40 cases in 8, in 18 cases in 9, in 26 cases in 10, in 22 cases in 11, and in the last 2 in 12, (giving an average of 9.1). In 46 of the possible strategies we get to the global optimum (cost=705), in 44 we reach the local minimum with cost=717, and in the remaining 30 the local minimum with cost=711. These results are to be compared to the 32 iterations required by the program in appendix A to reach the optimum. For Casey's 19-site example, we hand computed the algorithm for the search strategy obtained from the search factors (10, 11, 9, 8, 12, 7, 6, 1, 2, 5, 3, 4, 18, 19, 17, 14, 15, 16, 13 or, if condition 3 is used, 10, 11, 9, 8, 12, 7, 6, 1, 2, 18, 14, 16, 13). For Casey's "10% problem" we used 42 (30 with condition 3) node evaluations and got within 12% of the optimum cost. (We got 8, 10, 14 as opposed to 2, 10, 14). For the "20% problem" we used 23 iterations (17 if condition 3 is used) to get within 5% of the optimum (local minimum=10, 12). For the "30% problem" we used 23 node evaluations (17 with condition 3) and got within 1% of the optimum (local minimum = 10, 12). And finally, for the "40% problem" we used again 23 iterations (17) and obtained the optimum. In none of the above applications of our search algorithm was a node cost reevaluated. This seems to indicate that our strategy of going as far away as possible worked.

A.9 Conclusion

The current literature includes many approaches to file allocation optimization. For most of the current networks, the small number of sites makes it irrelevant which approach we take, as long as it suits our requirements. However, we have seen that we really mean small networks; i.e., 30 sites is big and 50 is tremendous. This follows from the exponential behavior of the problem.

We end up this appendix with no clear favorite. Whenever it is applicable Casey's algorithm seems to be a straightforward approach. Within his model we have built up a few conditions and search algorithms that offer some very attractive savings.

Casey's results are based upon an unrestricted environment; paradoxically this makes it somewhat restrictive. When we start imposing restrictions, such as limited local memory, limited bandwidth, etc., we find it hard to apply Casey's approach.

In Chu's zero-one integer programming method it is much easier to add restrictions, as mentioned above, but again we could say that Chu's model (like Casey's) is restricted with respect to time-varying parameters. Levin [L2] tried to solve this difficulty, but based his work upon Casey's; thus it has the same problems. No author that we know of has done anything to eliminate the static constraints from Chu's model.

As we shift from Casey's to Chu's model, the cost of obtaining a solution increases. In the light of our results, this might be a very important restriction.

Given that file allocation optimization for distributed data bases has been (for the most part) a mental exercise, none of the available optimization methods have paid attention to the details of distributing. This has led everybody to assume an environment which is essentially Johnson's model. No one has

considered the effect of including other schemes for ordering updates (see chapter VI where we study an alternate model), positive acknowledgement traffic, etc. Casey's simplification goes as far as to consider the flow of queries from user to file, but not any traffic subsequently generated (e.g., sending back response, consulting data at another site, etc.). We conclude that the existing literature is sufficient to give us an idea as to what we can use. But we first need to define our problem carefully and precisely before we can either use one of the existing methods or design a new one.

Appendix B. PROOF OF CONDITIONS 1, 2, AND 3 OF APPENDIX A

Proof of Condition 1.

Let's assume that there is a site A such that $Z_A < \lambda_A \min_{B:B\neq A} d_{AB}$, but an optimal allocation I does not include it. $(Z_A = \sum_{j=1}^{n} \Psi_j d'_{jA} + \sigma_A)$

$$C(I) = \sum_{x\in I} \sum_{j=1}^{n} \Psi_j d'_{jx} + \sum_{x\in I} \sigma_x + \sum_{j=1}^{n} \lambda_j \min_{x\in I} d_{jx}$$

Let $I' = I \cap \{A\}$. By hypothesis, $C(I) \leq C(I')$.

$$C(I') = \sum_{x\in I} \sum_{j=1}^{n} \Psi_j d'_{jx} + \sum_{j=1}^{n} \Psi_j d_{jA} + \sum_{x\in I} \sigma_x + \sigma_A + \sum_{j=1}^{n} \lambda_j \min_{x\in I} d_{jx} + \alpha,$$

where $\alpha = \sum_{j=1}^{n} \lambda_j (\min_{x\in I'} d_{jx} - \min_{x\in I} d_{jx})$. Clearly each term in α is nonpositive, since taking a minimum over a larger set can not increase that minimum. Furthermore, since for j = A the contribution to α is

$$-\lambda_A \min_{x\in I} d_{Ax}, \quad \alpha \leq -\lambda_A \min_{x\in I} d_{Ax}$$

and hence $\alpha \leq -\lambda_A \min_{B:B\neq A} d_{AB}$.

Thus, since

$$C(I') = C(I) + Z_A + \alpha, \text{ and}$$

given that $Z_A < \lambda_A \min_{B\neq A} d_{AB}$ and $\alpha \leq -\lambda_A \min_{B\neq A} d_{AB}$,

then $C(I') < C(I)$.

Contradiction!! A must be included.

Note. If a small nonzero local query cost is assumed, so that instead of $d_{AA} = 0$ we have $d_{AA} < \min_{B\neq A} d_{AB}$ for all A, then condition 1 becomes

$$\lambda_A(\min_{B\neq A} d_{AB} - d_{AA}) > Z_A,$$

and the proof above goes through with only minor changes.

Proof of Condition 2.

Given an allocation $I \neq 0$ with cost C_I, let $I' = I \cap \{A\}$.

Consequently

$$C(I') = C(I) + Z_A + \sum_{j=1}^{n} \lambda_j(\min_{x\in I'} d_{jx} - \min_{y\in I} d_{jy})$$

Then $C(I')$ will always be bigger than $C(I)$ if $Z_A > \sum_{j=1}^{n} \lambda_j(\min_{y\in I} d_{jy} - \min_{x\in I'} d_{jx})$

Now, making a term-by-term comparison, we find that

$$\sum_j \lambda_j(\max_C d_{jC} - d_{jA}) > \sum_j \lambda_j(\min_{y\in I} d_{jy} - \min_{x\in I'} d_{jx})$$

Thus, condition (2) is sufficient to imply that $C(I') > C(I)$.

If $I = 0$ the proof clearly will not go through. That is why we included the phrase "containing more than one site" in the statement of condition 2.

Proof of Condition 3.

The proof requires two preliminary lemmas.

Lemma 1.

$$\text{If } \sum_{j'} \lambda_{j'}(d_{j'A} - d_{j'B}) < Z_B \qquad \text{(B-a)}$$

then $C(I \cap A \cap B) > C(I \cap A)$.

Proof:

As in the preceding proofs,

$$C(I \cap B \cap A) - C(I \cap A) = Z_B + \sum_{j=1}^{n} \lambda_j(\min_{x\in I\cap A\cap B} d_{jx} - \min_{y\in I\cap A} d_{jy})$$

and $C(I \cap B \cap A) > C(I \cap A) \Leftrightarrow Z_B > \sum_j \lambda_j(\min_{y\in I\cap A} d_{jy} - \min_{x\in I\cap A\cap B} d_{jx})$ (B-b)

Condition (B-a) guarantees (B-b) if and only if

$$\sum_{j'} \lambda_{j'}(d_{j'A} - d_{j'B}) \geq \sum_{j=1}^{n} \lambda_j \left(\min_{y \in I \cap A} d_{jy} - \min_{x \in I \cap A \cap B} d_{jx} \right) \qquad \text{(B-c)}$$

Looking at (B-c) term by term, we have 3 cases:

1) $\min_{x \in I \cap A \cap B} d_{jx} \neq d_{jA}, d_{jB}$

In this case the right side of (B-c) is zero and the left is nonnegative.

2) $\min_{x \in I \cap A \cap B} d_{jx} = d_{jA}.$

In this case both sides are zero. (j does not satisfy the conditions on j'.)

3) $\min_{x \in I \cap A \cap B} d_{jx} = d_{jB}$

In this case the left side is no smaller than the right since $d_{jA} \geq \min_{y \in I \cap A} d_{jy}$,

and the second terms ($-\lambda_j d_{jB}$) are the same. We conclude that (B-c) always holds

and thus (B-a) is enough to imply that $C(I \cap A \cap B) > C(I \cap A)$.

Lemma 2.

$$\text{If } \sum_{j'} \lambda_{j'}(d_{j'A} - d_{j'B}) < Z_B - Z_A \qquad \text{(B-d)}$$

then $C(I \cap A) < C(I \cap B)$.

Proof:

$$C(I \cap B) - C(I \cap A) = Z_B - Z_A + \sum_{j=1}^{n} \lambda_j \left(\min_{x \in I \cap B} d_{jx} - \min_{y \in I \cap A} d_{jy} \right).$$

Therefore

$$C(I \cap B) > C(I \cap A) \Leftrightarrow Z_B - Z_A > \sum_{j=1}^{n} \left(\min_{y \in I \cap A} d_{jy} - \min_{x \in I \cap B} d_{jx} \right) \lambda_j.$$

Now if (B-d) holds, then it is only necessary to prove that

$$\sum_{j'} \lambda_{j'}(d_{j'A} - d_{j'B}) \geq \sum_{j=1}^{n} \lambda_j \left(\min_{y \in I \cap A} d_{jy} - \min_{x \in I \cap B} d_{jx} \right).$$

As in the proof of Lemma 1, we have 3 cases:

1) $\min_{x \in I \cap A \cap B} d_{jx} \neq d_{jA}, d_{jB}$

Then the right side is zero and the left site is nonnegative.

2) $\min_{x\ I \cap A \cap B} d_{jx} = d_{jA}$

Then the left side is zero and the right side is negative (or zero if $d_{jA} = d_{jB}$).

3) $\min_{x \in I \cap A \cap B} d_{jx} = d_{jB}$

Then the left side is at least as big as the right, by the same argument as for case 3) in Lemma 1.

Then $C(I \cap A) < C(I \cap B)$ if (B-d) holds.

Finally, we apply these lemmas to prove condition 3.

Since $Z_A \geq 0$,

if $\sum_{j'} \lambda_{j'}(d_{j'A} - d_{j'B}) < Z_B - Z_A$, then Lemma 1 and Lemma 2 hold.

Given two sites A,B the optimal allocation may include:

i) neither of them

ii) only one of them

iii) both of them

In case i) site B is not included.

In case ii) the inclusion of site B is nonoptimal since lemma 2 tells us that if we insert A for B we get a cheaper allocation.

In case iii) the inclusion of site B is nonoptimal because lemma 1 guarantees that if we exclude it we get something cheaper.

Therefore Site B should never be included; thus it is neighborly excluded by A.

Appendix C. PROGRAM TO SEARCH THE COST TREE

```
/* The following program was written in the algol-like C language
   of Bell Labs' UNIX (PDP-11) system. It is designed to imple-
   ment the preorder search of the modified cost tree (see fig.
   A.6-a) discussed in appendix A.*/
#define n 5                 /*test number*/
#define nplus1 6

/*      global variables      */
 double up[n][n],           /*update cost matrix*/
        qu[n][n],           /*query cost matrix*/
        sigma[n],           /*storage cost vector*/
        z[n],               /*cost of storing a copy. See appendix A.*/
        stkcost[n],         /*stack entry for allocation cost*/
        stkqucst[nplus1][n];/*stack entry to store query cost paid
                              in this allocation (first index) for the
                              query load of each host(second index)*/
 int    stkvalue[n],        /*stack entry to store number of host added
                              in this allocation*/
        level,              /*pointer to stack. Equiv. to tree level*/
        count,              /*number of allocations tried*/
        yesno,              /*0 if detailed listing is desired*/
        alloca[n];          /*contains a 1 in alloca[i] if host i+1 has a
                              copy of the file in current allocation*/

/*   main procedure     */
main ()
{int    minalloc[n],       /*equiv. to alloca for minimal alloc*/
        j;                  /*miscellanea*/
 extern double up[n][n], qu[n][n], sigma[n], z[n], stkcost[n],
             stkqucst[nplus1][n];
 extern int stkvalue[n],level,alloc[n],count,yesno;
 double minimum;            /*contains minimal current cost*/
 int    pop();

 /*       code         */
 getdata();    /*gets qu, up, and sigma. computes z*/
 printf("do you want detail ?yes(0) or no(1)\n");
 scanf("%d",&yesno);
 /* first some initialization */
 stkcost[0] = 999999;  /*cost of null allocation. we use 999999 */
                       /* as infinity.*/
 minimum = 999999;
 level = 0;            /*we start at the tree's root*/
 push(1);              /*evaluates alloca (1) and fills stk... [1] */

 /* main loop follows */
 while (level > 0) {
    if (stkcost[level] < stkcost[level - 1]) /*then we roll down
                                               i.e. its better*/
       { if (stkcost[level] < minimum)    /*then its a new minimum*/
```

```
                {minimum = stkcost[level];
                  for(j = 0; j < n; j = j+1) minalloc[j] = alloca[j];
                }
              if (stkvalue[level] < n)   /*then we can grow one level */
                 push (stkvalue[level] + 1);
              else /*can't grow more thru this branch, try next one*/
                 {j = pop();  if (j == n) j = pop();
                  push( j + 1);  /*we're in the next branch*/
                 }
           }
        else /*no optimum this way. we trim tree and go to next branch*/
           {j = pop();j = pop(); push(j + 1);  /*we're in next branch*/
           }
     } /*end of while and main loop */
     printf("minimum cost = %f for allocation = ", minimum);
     for (j = 0; j < n; j = j + 1)
        if (minalloc[j] == 1) printf("%d,",j + 1);
     printf("\n visited %d nodes",count);
    }/* end of main procedure */

    /*  procedure push  */
    push(site) int site;
    {/*We are given a site to be included in the allocation by adding it
       at location level+1 of the stack. Proper cost is computed and
       level is incremented by 1.*/
       extern double qu[n][n], stkqucst[nplus1][n], z[n], stkcost[n];
       extern int    level,stkvalue[n],yesno;
       double        cost;
       int           j;
       if (level < 0) return; /*this happens only at the end*/
       if (level == 0) cost=0.0;  /*no previous allocation*/
       else cost = stkcost[level];
       /*we now obtain effects of query transmission savings*/
       for (j = 0; j < n; j = j + 1)
          { if (level == 0)  /*no previous allocation */
               { cost = cost + qu[j][site - 1];
                 stkqucst[level+1][j] = qu[j][site - 1];
               }
             else /* we find out if we get some savings*/
               { if (stkqucst[level][j] > qu[j][site - 1])/*we do save*/
                    {cost = cost - stkqucst[level][j] + qu[j][site - 1];
                     stkqucst[level + 1][j] = qu[j][site - 1];
                    }
                 else  /*no savings, we keep previous valid cost*/
                    stkqucst[level + 1][j] = stkqucst[level][j];
               }
          }
       /*we now know query savings. we finish our job next*/
       level = level + 1;
       stkcost[level]=cost+z[site-1]; /*we add update and storage cost*/
       stkvalue[level] = site;
       alloca[site - 1] = 1;
       count = count + 1;
       if (yesno == 0) /*we print detail*/
          printf("we add site %d at level %d and obtain cost %f\n",
                site, level, stkcost[level]);
    }
```

```
/*  procedure pop    */
int pop()
 { extern int stkvalue[n],level,alloc[n],yesno;
   if (yesno == 0) printf("pop\n");
   alloca[stkvalue[level] - 1] = 0; /*we deallocate*/
   level = level - 1;
   return(stkvalue[level + 1]);
 }

/*  procedure getdata  */
getdata()
{
   extern double up[n][n],qu[n][n],sigma[n],z[n];
   /*  Here we load data for up, sigma, and qu. Then we compute z.*/
   /*  The details of this procedure are of no particular interest*/
   /*  and are left out from this listing.                        */
}
```

Appendix D. PROOF OF CASEY'S THEOREM FOR A PRIMARY SCHEME

We now prove the following lemma, analogous to Casey's lemma [C1, p. 620] for a non-primary model. Just as in Casey's model, this lemma leads straightforwardly to the fact that costs are monotonic non-increasing along paths to the minimum. (See chapter IV.)

Without loss of generality we will assume that the primary is at site 1. Sites 2 and 3 are any other two sites. We have to prove that if allocation $I = \{1,2,\ldots,r\}$ is cheaper than allocation $I - \{2\}$ and $I - \{3\}$ then its cheaper than $I - \{2,3\}$. The notation $I - \{k\}$ has the same meaning as in chapter VI, i.e., allocation I excluding site k.

Lemma.

If $C(I) \leq C(I - \{k\})$, $k = 2,3$;

then

$$C(I - \{k\}) \leq C(I - \{2,3\}) \qquad k = 2,3.$$

Proof.

Let $X_k = C(I - \{k\}) - C(I)$, and

$$Y_k = C(I - \{2,3\}) - C(I - \{k\})$$

We wish to show that if $X_2 \geq 0$ and $X_3 \geq 0$, then $Y_2 \geq 0$ and $Y_3 \geq 0$.

This will be true if $Y_3 - X_2 \geq 0$ and $Y_2 - X_3 \geq 0$.

Now $X_2 = -\sigma_2 - \Psi d_{12} + \sum_j \lambda_j \left(\min_{k \in I-\{2\}} d_{jk} - \min_{k \in I} d_{jk} \right)$

$$Y_3 = -\sigma_2 - \Psi d_{12} + \sum_j \lambda_j \left(\min_{k \in I-\{2,3\}} d_{jk} - \min_{k \in I-\{3\}} d_{jk} \right)$$

So $Y_3 - X_2 = \sum_j \lambda_j \left(\min_{k \in I-\{2,3\}} d_{jk} - \min_{k \in I-\{3\}} d_{jk} - \min_{k \in I-\{2\}} d_{jk} + \min_{k \in I} d_{jk} \right)$.

The remainder of the proof follows exactly Casey's proof; we will not copy it here. Essentially, one sees easily that the quantity

in parentheses is positive for all j; and the inequality $Y_2 - X_3 \geq 0$ follows by permuting indices.

The path-searching program in appendix C will therefore still work. The primary is fixed a priori, and allocations not including the primary are assigned infinite cost. To find the best (on the basis of cost) site for the primary, we may do the optimization n times, each time assuming that a different site is the primary. Then we choose the optimum site as the one giving the minimum of all minima. This approach expands our problem by a factor of n/2; i.e., we have n problems of size roughly half as big as before.

LIST OF REFERENCES

A1. Alsberg, P.A.
"Distributed Processing on the ARPA Network - Measurements of the Cost and Performance Tradeoffs for Numerical Tasks," Proceedings of the Eight International Conf. on System Science, pp. 19-24, 1975.

A2. Alsberg, P.A.
"Space and Time Savings Through Large Data Base Compression and Dynamic Restructuring," Proceedings of the IEEE, Vol. 63, No. 8, August 1975.

A3. Alsberg, P.A., Belford, G.G., Bunch, S.R., Day, J.D., Grapa, E., Healy, D.C., McCauley, E.J., and Willcox, D.A.
"Synchronization and Deadlock," CAC Doc. 185 (CCTC-WAD Doc. 6503), Center for Advanced Computation, University of Illinois at Urbana-Champaign, March 1976.

A4. Alsberg, P.A., Belford, G.G., Day, J.D. and Grapa, E.
"Multi-copy Resiliency Techniques," CAC Doc. 202 (CCTC-WAD Doc. 6505), Center for Advanced Computation, University of Illinois at Urbana-Champaign, May 1976.

A5. Alsberg, P.A. and Day, J.D.
"A Principle for Resilient Sharing of Distributed Resources," To be presented in the 2nd International Conf. on Software Engineering, October 1976.

B1. Belford, G.G., Day, J.D., Sluizer, S., and Wilcox, D.A.
"Initial Mathematical Model Report," CAC Doc. 169 (JTSA Doc. 5511) Center for Advanced Computation, University of Illinois at Urbana-Champaign, August 1975.

B2. Belford, G.G., Schwartz, P.M., and Sluizer, S.
"The Effect of Backup Strategy on Data Base Availability," CAC Doc. 181 (CCTC-WAD Doc. 6501), Center for Advanced Computation, University of Illinois at Urbana-Champaign, February 1976.

B3. Belford, G.G.
"Optimization Problems in Distributed Data Managment," CAC Doc. 197 (CCTC-WAD Doc. 6504), Center for Advanced Computation, University of Illinois at Urbana-Champaign, May 1976.

B4. Belford, G.G., Day, J.D., Grapa, E., and Schwartz, P.M.
"Network File Allocation," CAC Doc. 203 (CCTC-WAD Doc. 6506), Center for Advanced Computation, University of Illinois at Urbana-Champaign, August 1976.

B5. Bunch, S.R.
"Automated Backup," in Preliminary Research Study Report, CAC Doc. 162 (JTSA Doc. 5509), Center for Advanced Computation, University of Illinois at Urbana-Champaign, May 1975.

C1. Casey, R.G.
"Allocation of Copies of a File in an Information Network," AFIPS Conf. Proceedings, Vol. 40, pp. 617-625, 1972.

C2. Chandy, K.M. and Hewes, J.E.
"File Allocation in Distributed Systems," Proceedings International Symp. on Comp. Performance Modeling, Measurement and Evaluation, pp. 10-13, March 1976.

C3. Chang, S.
"Data Base Decomposition in a Hierarchical Computer System," International Conf. on Management Data, ACM SIGMOD, pp. 48-53, May 1975.

C4. Chu, W.W.
"Optimal File Allocation in a Multi-computer Information System," IEEE Transactions on Computers, Vol. C-18, No. 10, pp. 885-889, October 1969.

C5. Chu, W.W.
"Optimal File Allocation in a Computer Network," in Computer-Communications Networks, N. Abramson and F. Kuo (Eds.), Prentice-Hall, Englewood Cliffs, N.J., 1973.

D1. Day, J.D.
"Resilient Protocols for Computer Networks," in Preliminary Research Study Report, CAC Doc. 162 (JTSA Doc. 5509), Center for Advanced Computation, University of Illinois at Urbana-Champaign, May 1975.

D2. Day, J.D. and Belford, G.G.
"A Cost Model for Data Distribution," CAC Doc. 179 (JTSA Doc. 5514), Center for Advanced Computation, University of Illinois at Urbana-Champaign, November 1975.

D3. Dijkstra, E.W.
"Co-operating Sequential Processes," Programming Languages, F. Genuys, (ed.), New York, Academic Press, pp. 43-112, 1968.

E1. Eswaran, K.P.
"Placement of Records in a File and File Allocation in a Computer Network," IFIP 74, Amsterdam, pp. 304-307, 1974.

F1. Frank, H., Kahn, R.E., and Kleinrock, L.
"Computer Communication Network Design - Experience with Theory and Practice," Spring Joint Computer Conf., AFIPS Conf. Proceedings, Vol. 40, pp. 255-270, 1972.

G1. Grapa, E. and Belford, G.G.
"Some Theorems to Aid in Solving the File Allocation Problem," Submitted to CACM.

H1. Harris, B.
"Theory of Probability," Addison-Wesley, 1966.

J1. Johnson, P.R. and Beeler, M.
"Notes on Distributed Data Bases," Draft Report, available from the authors (Bolt, Beranek, and Newman, Inc., Cambridge, Mass.), 1974.

J2. Johnson, P.R. and Thomas, R.H.
"The Maintenance of Duplicate Databases," RFC #677, NIC #31507, Jan. 1975. (Available from ARPA Network Information Center, Stanford Research Institute, Augmentation Research Center, Menlo Park, CA.)

K1. Kleinrock, L.
"Analytic and Simulation Methods in Computer Network Design," Spring Joint Computer Conf., Atlantic City, N.J., AFIPS Conf. Proceedings, 36, pp. 569-578, 1970.

K2. Kleinrock, L.
"Scheduling, Queueing, and Delays in Time-Shared Systems and Computer Networks," in Computer Communication Networks, N. Abramson and F.F. Kuo (Eds.), Prentice-Hall Englewood Cliffs, N.J., pp. 95-141, 1973.

L1. Lamport, L.
"Time, Clocks and the Ordering of Events in a Distributed System," Massachusetts Computer Associates, Inc., March 1976.

L2. Levin, K.D.
"Organizing Distributed Data Bases in Computer Networks," Ph.D. Dissertation, University of Pennsylvania, 1974.

M1. Martin, J.
"Security, Accuracy and Privacy in Computer Systems," Prentice-Hall, Inc., 1973.

N1. Naylor, W. and Opderbeck, H.
"Mean Round-Trip Times in the ARPANET," RFC #619, NIC #21990, Network Measurement Group Note #19, NIC #21791, March 1974.

S1. Spinetto, R.D.
"A Facility Location Problem," SIAM Review 18, pp. 294-295, 1976.

U1. Urano, Y., Ono, K., and Inoue, S.
"Optimal Design of Distributed Networks," ICCC Stockholm, pp. 413-420, 1974.

LIOGRAPHIC DATA ET	1. Report No. UIUCDCS-R-76-831	2.	3. Recipient's Accession No.
itle and Subtitle CHARACTERIZATION OF A DISTRIBUTED DATA BASE SYSTEM			5. Report Date October 1976 6.
uthor(s) Enrique Grapa			8. Performing Organization Rept. No. UIUCDCS-R-76-831
erforming Organization Name and Address Department of Computer Science University of Illinois at Urbana-Champaign Urbana, Illinois 61801			10. Project/Task/Work Unit No. 11. Contract/Grant No. DCA100-76-C-0088
Sponsoring Organization Name and Address Command and Control Technical Center WWMCCS - ADP - Directorate 11440 Isaak Newton Square, North Reston, Virginia 22090			13. Type of Report & Period Covered Ph.D. Thesis 14.

Supplementary Notes

Abstracts
A distributed data base system forms a very attractive solution to various of the ual data base problems in a computer network. Availability, response time, and rational cost savings make multiple copy (distributed) data base systems not only ractive but economically appealing.

The majority of the researchers involved in related work have adopted a decentral-d inter-copy synchronization scheme which turns out to be very restricted. They ume that whenever a data user wants to perform an update he will send it to every puter host which possesses a copy of the data base. Given that the order in which ates are applied does matter, some means of synchronizing the update application at hosts must be provided. It turns out that the available models which provide de-tralized synchronization can efficiently manipulate only a very restricted set of ate operations.

(ntinued on next page)

ey Words and Document Analysis. 17a. Descriptors

tributed Data Bases, File Allocation Distributed Networks, Computer Networks

dentifiers/Open-Ended Terms

COSATI Field/Group

ailability Statement	19. Security Class (This Report)	21. No. of Pages
Release Unlimited	UNCLASSIFIED	179
	20. Security Class (This Page) UNCLASSIFIED	22. Price --

NTIS-35 (10-70)

USCOMM-DC 40329-P71

(continuation of No. 16 from the preceding page)

Three models for update synchronization are presented and extensively studied. The first is Johnson's model, which basically assumes that updates are handled in the decentralized fashion of our previous description. The second is Bunch's model, which introduces the concept of a centralized scheme. In this scheme, all updates are sent to a primary host which in turn broadcasts them to all backup hosts that hold a data base copy. Finally, we introduce the Reservation Center model developed by the author. The Reservation Center model combines various centralized and decentralized concepts.

The major flaws of the models are discussed and extensions are presented to cover them. The broadcasting model, an extension of Bunch's model, is presented as a prototype of a workable generalized distributed data base system.

After our discussion, it seems likely that a major reorientation should be made in the related literature to cover the centralized synchronization schemes. We have done so with Casey's file allocation model, with minimal consequences to the general applicability of his work.

During our study of the file allocation problem we have made some contributions in the form of a priori conditions for the inclusion or exclusion of a host in an optimal file allocation. These results are presented in an appendix.